Lecture Summary

SPCA

Simone Goggia

Table of Contents

1 Introduction 2
2 Introduction to C 2
3 Representing C Integers 4
4 Pointers 5
5 Dynamic Memory Allocation 7
6 C Wrap-Up 8
7 Basic x86 Architecture 10
8 Compiling C Control Flow 12
9 Compiling C Data Structures 16
10 Code Vulnerabilities 18
11 Memory Allocation 18
12 Linking 24
13 Floating Point 26
14 Optimizing Compilers 29
15 Architecture and Optimization 30
16 Caches 32
17 Exceptions 34
18 Virtual Memory 36
19 Multiprocessing 40
20 Devices 44

Info

There is no claim for completeness. All warranties are disclaimed. Creative Commons Attribution-Noncommercial 3.0 Unported license.

Study Part

Disclaimer: I have already attended course 252-0062-00L "Operating Systems and Networks" by Prof. Hoefler and Prof. Perrig. Due to that some elements in this summary might not be discussed extensively. See also: http://studysheets.ch/sheets/operating-systems/download.

Since writing code is rather tedious in Microsoft Word and I am really in favor of saving paper, the following variable declarations may be assumed:[1]

```
int i; char c; float f; double d; // etc.
```

1 Introduction[2]

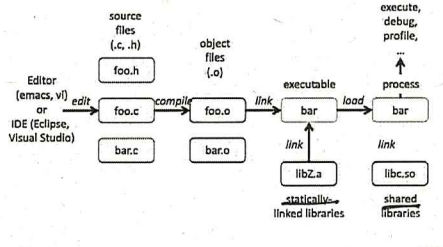
Five important realities to always keep in mind:

1. ints are not integers – floats are not real numbers. E.g. $i^2 \geq 0$ only holds for floats, not necessarily for ints. While computer arithmetic doesn't generate random values not all "usual" mathematical properties may be assumed. Integer operations stratify properties of rings, floating point operations satisfy ordering properties.
2. You've got to know assembly. It is the key to understanding the machine-level execution model.
3. Memory matters – RAM is an unrealistic abstraction. Memory is not unbounded, is the source of pernicious bugs, and memory performance is not uniform.
4. There's much more to performance than asymptotic complexity. Not only do constant factors but you also have to understand the system to optimize performance.
5. Computers don't just execute programs. I/O is critical to reliability and performance. Additionally there's network communication which is the source for many system-level issues.

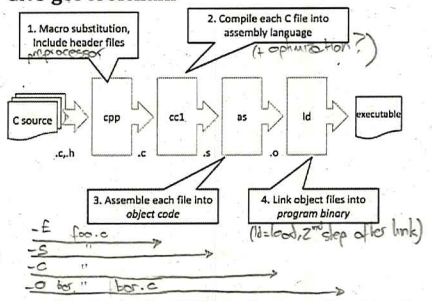
2 Introduction to C

Contrary to languages such as C# or Eiffel, C is very fast, is close to the metal, and uses a powerful **macro pre-processor** (cpp). The cpp performs string and file substitution and conditional compilation. It is the choice for OS developers, embedded systems, speed fanatics, and authors of security exploits. On the other hand, C lacks OOP features, a lot of built-in types, and exceptions. Furthermore it doesn't have automatic memory management but it has pointers which offer **direct access to memory** addresses. A return value of 0 indicates everything went okay.

Workflow



GNU gcc toolchain



[1] By personal preference, I use Source Code Pro as monospace font. In the final PDF the fonts are (or at least should be) embedded. If you have problems viewing this file, please install the font by downloading it for free from Adobe on GitHub: github.com/adobe-fonts/source-code-pro

[2] The heading numbers [of level 2] correspond to the chapter/lecture numbers by Prof. Roscoe and the subheadings correspond to the different topics outlined in each lecture. *Exception:* Chapter 1

Control flow in C

Just like C# etc. C has if/else, switch, return, for, while, do/while, break, continue, goto statements. While goto is highly controversial, it does have a purpose in e.g. OS code where cleanup needs to be done, e.g. a function has to perform an operation and needs to do three things before it can do its main purpose. If any of these three "things" fail, they need to be cleaned up. To do so, their cleanups are written in reverse order after the main part and gotos are used to jump there. C relies on the main() function, it starts off the whole program.

```
/* program to print arguments from command
line */
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
  printf("argc = %d\n\n", argc);
  for (i=0; i<argc; ++i) {
    printf("argv[%d]: %s\n",i, argv[i]);
  }
  return 0;
}
```

Basic types in C

Declarations within a block are local to that block whereas declarations outside of a block are declared in the entire program. Static inside a block persists between calls, outside blocks it is limited to the file.

C data type	Intel x86-64
char	1
short	2
int	4
long	8
~~long long~~	8
float	4
double	8
long double	10/16

Integers are signed by default; "signed" and "unsigned" can be used to clarify. Types have different sizes on different architectures, the right-hand table lists the sizes for Intel x86-64. Rules for arithmetic on integers and floats are complex since they involve implicit and explicit (casts) conversions.

Booleans are just integers (0::`false` and non-0::`true`) and the "`!`" operator turns anything non-zero into 0 and vice-versa. Support for a new `bool` type was added in C99 yet is completely optional. Statements in C are also an expression can be useful for e.g. file-exists calls.

`void` is a type and doesn't have a value. It is used as an untyped pointer to [raw] memory and for functions with no return value (aka procedures).

Operators

C supports a wide array of operators including shifts, the ternary if-else operator, bitwise operator, and (arithmetic) assignment operators. Additionally, the following are considered operators for operating on pointers: *, &, (type), sizeof. In- and decrements come in pre and post flavor and differ in what value the variable being incremented has when accessing it. This works for scalar types and pointers. Casting is available for most types.

Arrays in C

An array is a finite vector of variables which are all of the same type and indices are zero-based. The compiler does not perform bound checking. To initialize an array, different methods are available.

```
int a[3] = {3, 7, 9};
a[0] = 3,
a[1] = 7,
a[2] = 9
```

```
float list[100] = {};
list[0]=0.0,
…,
list[99]=0.0
```

```
int a[3][3] = {
{ 1, 2, 3},
{ 4, 5, 6},
{ 7, 8, 9},
};
```


Strings are an array of chars in C, terminated with null `\0`. Henceforth, `char str[6] = {'h','e','l','l','o','\0'};` is equivalent to `char str[6] = "hello";`. Yet C does provide a lot of library functions to operate on strings.

3 Representing C Integers

Bit-wise operators treat arguments as bit vectors while logic operators always return 0 or 1 (while treating 0 as false, and anything else as true) and may terminate early. To avoid null pointer access, the following trick can be used: `p && *p`. The bitwise operators have the following meanings as vector operations: "&"→ intersection, "|"→ union, "^"→ symmetric difference, "~"→ complement. Shift operations the following properties:

Left shift: x << y	Right shift: x >> y	Undefined behavior
– Shift bit-vector x left y positions – Throw away extra bits on the left – Fill with 0s on the right	– Shift bit-vector x right y positions – Throw away extra bits on the right – Logical shift: fill with 0s on the left – Arithmetic shift: replicate MSB on the right	– Shift amount < 0 – Sift amount ⩾ word size

Integer ranges

	Unsigned	Two's complement
Conversion	$B2U(x) = \sum_{i=0}^{w-i} x_i \cdot 2^i$	$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$
Min **$w = 16$**	UMin = 0 = 000 … 0	TMin = -2^{w-1} = 100 … 0 −32768 = 0x8000
Max **$w = 16$**	UMax = $2^w - 1$ = 111 … 1 65535 = 0xFFFF	TMax = $2^{w-1} - 1$ = 011 … 1 32767 = 0x7FFF
-1	N/A	0xFFFF
0	0x0000	
Observations	|TMin| = TMax + 1, UMax = 2 · TMax + 1; 2's complement: $\sim x + 1 = -x$; $\sim x + x$ = 111 … 1 = −1	

Constants in C are considered to be singed integers. Casting between signed and unsigned is possible using "(int)" and "(unsigned)", respectively. Casting can also happen implicitly. When mixing signed and unsigned numbers in an expression, however, singed values are implicitly cast to unsigned. Sign extension works by copying the MSB. C automatically performs sign extension for signed values.

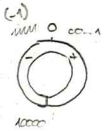
Integer addition and subtraction in C

$$s = UAdd_w(u, v,) = u + v \bmod 2^w = \begin{cases} u + v, & u + v < 2^w \\ u + v - 2^w, & u + v \geq 2^w \end{cases}$$

The standard **unsigned addition** function ignores the carry output (a w bit operand would result in a $w + 1$ bit number) and thus implements modular arithmetic; it wraps around when the true sum is $\geq 2^w$. This operation forms an Abelian group: it is closed under addition, commutative, associative, 0 is the additive identity, and each element has an additive inverse.

$$TAdd_w(u,v) = \begin{cases} u+v+2^w, & \overbrace{u+v < TMin_w}^{\text{negative overflow}} \\ u+v, & TMin_w \le u+v \le TMax_w \\ u+v-2^w, & \underbrace{TMax < u+v}_{\text{positive overflow}} \end{cases}$$

Unsigned and signed addition have the same bit-level behavior in C. Performing **two's complement addition** also requires $w+1$ bits, and it then drops off the MSB and treats the remaining bits as a two's complement integer. When rapping around it behaves as follows: if the sum is $\geq 2^{w-1}$ it becomes negative (at most once) and if the sum is $< -2^{w-1}$ it becomes positive (at most once). Addition in 2's complement forms a group. The group is isomorphic to unsigneds in unsigned addition.



↳ associativity $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
identity $e \cdot a = a \cdot e = a$
invertibility $a \cdot \hat{a} = \hat{a} \cdot a = e$
closure (all is contained)

Integer multiplication in C

	Unsigned (up to 2^w bits)	2's complement min (up to 2^{w-1} bits)	2's complement max (up to 2^w bits, but only for $(TMin_w)^2$)
Range	$0 \le x \cdot y \le (2^w-1)^2 = 2^{2w} - 2^{w+1} + 1$	$x \cdot y \ge (-2^{w-1}) \cdot (2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$	$x \cdot y \le (2^{w-1})^2 = 2^{2w-2}$

w = #bits

use min, which has max abs value

Unsigned multiplication produces a $2w$ bit results but discards w bits and thus implement modular arithmetic. Together with unsinged addition if forms a commutative ring: addition is a commutative group, it is close under multiplication, it is commutative and associative, 1 is the multiplicative identity, and multiplication distributes over addition.

Signed multiplication produces a $2w$ bit results but discards w bits. It is again isomorphic to unsigned multiplication and addition and both of them are isomorphic to ring of integers mod 2^w.

↳ generalize $+, *$

Integer multiplication and division using shifts

$u \ll k$ is equivalent to $u \cdot 2^k$, in both signed and unsigned representations. $u \gg k$ is the same as $\lfloor u/2^k \rfloor$ and uses a logical shift for unsigned numbers and an arithmetic shift for signed numbers (in that case, it also rounds in the wrong direction when $u < 0$. To get a correct quotient of a negative number, the following can be used $\lfloor (x + 2^k - 1)/2^k \rfloor \Leftrightarrow (x + (1 \ll k) - 1) \gg k$.

floor

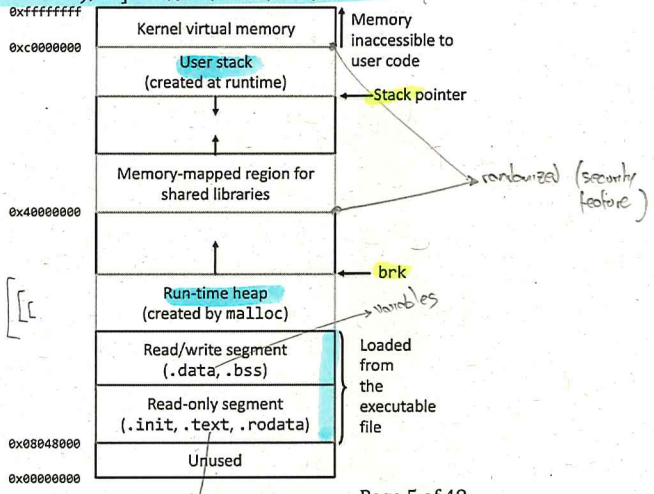
↳ es $\lfloor \frac{-7}{2} \rfloor = -4$ $1001 \Rightarrow 1100$ $-7 \Rightarrow -8$

4 Pointers

The stack

To support **recursion**, code must be reentrant, meaning there are multiple simultaneous instantiations of a single procedure. And the stack is where all arguments, local variables, and return pointers are saved for the time between when a routine is called and when it returns. To ensure stack discipline, the callee returns before the caller does. The stack is allocated in frames. The stack grows **downwards**.

caller callee



segmentation fault = dereference NULL ptr

VM

Pointers in C

`&x` produces the virtual address where the value of x is stored.[3] **A pointer is a variable which contains a memory address** and points to somewhere in the process' virtual address space. **Dereferencing** a pointer is access the memory referred to by a pointer. **NULL** is a guaranteed-to-be-invalid memory location and its type is `void * (0x00000000)`. Ayn attempt to dereference a null pointer leads to a segmentation fault.

```
// declare a pointer
type *name;
// declare + initialize a pointer
type *name = address;

// dereference a pointer
v = *pointer;
// dereference / assign
*pointer = value;
```

int v = 5
int *ptr = &v
*ptr = 6 ⇒ v is assigned 6

As a security feature, the **address space is randomized**. Linux randomizes the base of the stack and the locations of the shared libraries. This makes debugging more challenging.

Box-and-arrow diagrams

Omitted.

Pointer arithmetic

it's not a function; but an operator

You can perform arithmetic operations on pointers. These operations respect the size of `(sizeof(type), sizeof(value);` is evaluated at compile time). E.g. increasing a `char *` increases the address by one byte while increasing an `int *` increases the address by four bytes.

Arrays and pointers

int a[10] //unless
1) a = &a
2)
3) sizeof(a) = 10*sizeof(int)

An array name is an **expression** and is treated as a pointer to the first element of the array[4] *unless* (1) the array's address is taken with a `&`, (2) the array is a string literal initializer, or (3) the array is an operand of `sizeof()`.An array name as a function parameter is a pointer.[5] Arrays can't be renamed (compile-time error) but when referring to them as a pointer, it's possible.

Passing by reference

you don't modify the original

By default, C passes arguments by value thus giving the callee a **copy** of the value. This implies the callee cannot modify the caller's copy. When passing by reference, the callee still receives a copy of the argument, but now it is **pointer** of which the value points to the variable in the scope of the caller thus allowing the callee to **modify** the variable in the scope of the caller.

```
char *strcpy(char *dest, char *src)
{
    char *r = dest;
    while(*dest++ = *src++); // Mistakes the copying
    return r;
}
```

when does it return false?
@last char (\0)

- Strings are arrays of characters terminated by null bytes \0
- Assignment is an expression, not a statement
- Non-zero values evaluate to true, zero evaluates to false
- Post-increment operators bind more tightly than pointer dereference
- A semicolon is statement terminator, not a separator

dangerous

assert(bool) : if evaluates to false, write exor & abort

[3] `printf("x is at %p\n", &x);`
[4] The compiler rewrites `A[i]` always to `*(A+i)` ← increase based on type
[5] This is how functions are converted to pointers.

Declaration	Meaning
`int *p`	`p` is a pointer to `int`
`int *p[13]`	`p` is an `array[13]` of pointer to `int`
`int *(p[13])`	`p` is an `array[13]` of pointer to `int`
`int **p`	`p` is a pointer to a pointer to an `int`
`int (*p)[13]`	`p` is a pointer to an `array[13]` of `int`
`int *f()`	`f` is a function returning a pointer to `int`
`int (*f)()`	`f` is a pointer to a function returning `int`
`int (*(*f())[13])()`	`f` is a function returning `ptr` to an `array[13]` of pointers to functions returning `int`
`int (*(*x[3])())[5]`	`x` is an array[3] of pointers to functions returning pointers to `array[5]` of `int`s

5 Dynamic Memory Allocation

A global variable is **statically** allocated when the program is loaded and deallocated when it exits. Variables within functions are **automatically** allocated when the function is called and deallocated when the function returns. When there is a need for more memory which persist across multiple calls, is too big for the stack, or the required size isn't known to the caller, **dynamically** allocated memory is used. The program explicitly requests a new block of memory which persists until the code explicitly deallocates it.[6]

The C memory API

`malloc()` allocates a block of memory of the given size and returns a pointer to the first byte of that memory (and NULL if the memory cannot be allocated). The memory should be assumed to contain garbage. To calculate the size needed, `sizeof()` is typically used. `calloc()` behaves similarly except it takes 2 parameters and then multiplies them and it zeroes the memory out, making it a bit slower but also more readable and less error-prone.

```
// declared in stdlib.h
typedef unsigned int size_t;
void *malloc(size_t sz);
```

Deallocation is done using `free()` which releases the memory at the pointer. To do so, it has to point to the first byte of the allocated memory and it is a good practice to NULL the pointer after freeing the memory.

```
// declared in stdlib.h
void free(void *);
```

While allocations have a **fixed** size, memory can be **reallocated** to change the size of the block using `realloc()`. This operation most likely will copy the data to a new location and thus the new address returned has to be used.

`size_t` is an unsigned integer of some size and is also the return type of `sizeof()`. It is large enough to hold the size of the largest possible array in memory which makes it a suitable type to be used to store a pointer. `ptrdiff_t` is also an unsigned integer and is the result of subtracting two pointers. It is used for array loops, size calculations etc.

Managing the heap

The heap ("free store") is a large pool of unused memory which is used for dynamically allocated data structures. To keep track of that memory, `malloc()` maintains **bookkeeping data** of allocated blocks in the heap. **Memory leaks** happen when code doesn't deallocate memory which is no longer used. As an implication, the memory footprint of that program will keep growing which is

[6] Or it is collected by the garbage collector, a feature lacking in C.

often really bad. *Note:* garbage-collected languages are not memory-leak-prone, they're just much less likely.

Structures and unions

A `struct` is a C type which contains a set of fields and is comparable to class but it lacks methods and constructors. Instances can be allocated on the stack or on the heap. To refer to fields, a `.` is used and `->`[7] refers to field through a pointer to a `struct`. When copying by **assignment**, the entire contents are copied which is for example what happens when using them as arguments for a function (to pass by reference, a pointer to it is passed). Of course, you can also return a `struct`.

```
// "list_el" is the structure tag
struct list_el {
    unsigned long val;
    struct list_el *next;
};
// the struct keyword is needed
// to refer to a tag
// this also applies to unions .
struct list_el my_list;
```

Unions are like `struct`s and are also accessed as such, but they only hold *one of a set* of alternative values (but they do not check which value is correct).

Type definitions

A `typedef` introduces a new definition or rather a new name for a type. They can be used to build up declarations in an easily understandable fashion.

```
typedef unsigned uint32_t;
uint32_t ui;
…
typedef int **myptr;
int *p;
myptr mp = &p;
…
typedef struct skbuf skbuf_t;
skbuf_t *sptr;
```

```
// x is an array of 3 elements,
// each of which is a pointer to
// a function returning an
// array of 5 ints
// instead of int (*(*x[3])())[5]

typedef int fiveints[5];
typedef fiveints* p5i;
typedef p5i (*f_of_p5is)();
f_of_p5is x[3];
```

Dynamic data structures

Omitted.[8]

Generic data structures

Omitted.[9]

6 C Wrap-Up

The C preprocessor

As aforementioned, C has a powerful preprocessor. One usage is to include **header files** inline in the source code which is essentially a basic mechanism for defining APIs. Double-quotes are for local headers, greater/smaller-than signs are used for system headers. The cpp also supports **macro definitions** which work as a token-based

```
#include <file1.h>
#include "file2.h"
#define FOO BAZ
#define BAR(x) (x+3)
#undef FOO
#if expression
#elif expression
#ifdef FOO
#else
#endif
#ifndef BAR
```

[7] Which (`p->x`) is shorthand for (`*p`).`x`

[8] For an example on how to implement a singly-linked list, please see slides 35 – 40.

[9] For an example on how to implement a generic linked list, please see slides 42 – 44.

macro substitution. Furthermore, there is also support for conditionals. Since semicolons are a null statement in C, it has to be "swallowed" in macro definitions which is done by using backslashes.

Modularity

A function **declaration** says something exists somewhere ("prototype") while a function **definition** says what it is ("code"). C deals with **compilation units** which consist of a C file plus everything it includes. Declaration can be annotated with `extern` (definition is somewhere else, either in this compilation unit or another) or `static` (definition (also `static`) is in this compilation unit, and can't be seen outside it). The same also applies to global variables which are also declared. A **module** is a self-contained piece of a larger program. It consist of externally visible (aka interface) parts (functions to be invoked, `typedefs`, global variables, cpp macros) and internal parts (internal function, types, global variables). A C **header file** is used to specify interfaces. Clients include the header file (`foo.h`) which contains no definitions but only external declarations. The implementation is typically in `foo.c` (which also includes `foo.h`) and doesn't contain any external declarations but only definitions and internal declarations.

cpp boilerplate ensures file contents only appear once; never `#include` a .c file

```
#ifndef __FILE_H_
#define __FILE_H_
… (declarations, macros)
#endif // __FILE_H_
```

Function pointers

In this code: `int (*func)(int *, char);`, `func` is a pointer to a function which takes two arguments, a pointer to `int` and a `char`, and returns an `int`. This can be used with `typedefs`, just like any type, and is the basis for lots of techniques in systems code.

Assertions

```
assert( <scalar expression> );
```

Assertions are evaluated at **runtime** and if it evaluates to true, nothing happens, otherwise it prints an error message and the program aborts (core dump). When compiling with `-DNDEBUG`, assertions are removed. Assertions are macros and shouldn't contain side-effects. *They are for programmers to find bugs, not for programs to detect errors.*

goto

The `goto` construct is almost never a good idea, even though some argue on performance grounds. It can, however, be used for early termination of multiple loops and to cleanup nested code. It is used for **recovery code** where the code performs a sequence of operations and any one can fail and if it fails, all previous operations must be undone. A typical example is `mallocing` a sequence of buffers for data.[10]

setjmp() and longjmp()

`setjmp()` saves the current stack state in `env` and returns 0. `longjmp()` causes *another* return to the point saved by env. The new `return`, returns `val` (or 1 if `val` is 0). This can only be done once for each `setjmp()`. It is invalid if the function containing the `setjmp` returns.

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Coroutines

An example where coroutines can be used, is a decompression algorithm with a decompressor (which runs until it has a character limit) and a parser (which continues where it previously left

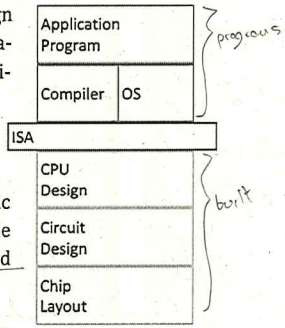
[10] This code is often auto-generated.

off, processes new characters and runs until it needs a new one, and then calls back to the decompressor).

7 Basic x86 Architecture

What is an instruction set architecture?

An **architecture** (also ISA) describes the parts of a processor design which is relevant to writing assembly code, such as instruct set specification, registers. A **microarchitecture** is an implementation of said architecture (cache sizes, frequency).



CISC stands for Complex Instruction Set Computer. It is **stack-oriented**; the stack is used to pass arguments (which saves the program counter), providing explicit push and pop instructions. Arithemetic instructions can access memory and condition codes are used as a side effect of artihmetic and logic instructions. The philosophy is to add instructions to perform typical programming tasks. x86 is CISC.

RISC stands for Reduced Instruction Set Computer. The instructions are fewer and simpler which might result in more instructions yet they can be executed on small and fast hardware. The instruction set is **register-oriented** which are quite numerous and are used for arguments, return pointers, and temproatries. Only load and store instructions can access memory and there are no condition codes (test instructions return 0/1 in a register). MIPS is RISC and is motivated by "simpler is faster"

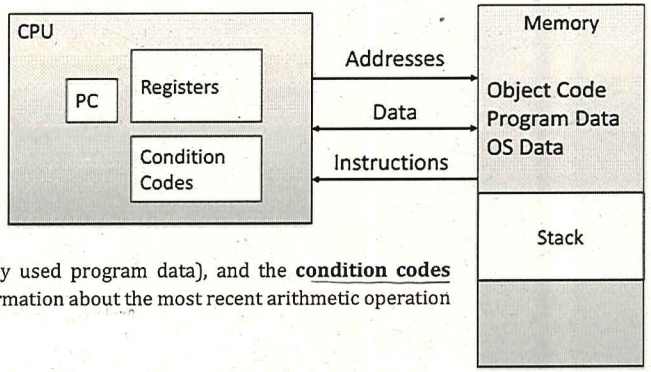
While there is still an ongoing debate between CISC (easy for compiler, fewer code bytes) and RISC (better for compiler optimization, make it run fast with simple chip design), currently RISC is still a sensible choice for embedded processors while the ISA choice is not a technical issue on desktop processors.

A bit of x86 history

Omitted.[11]

Basics of machine code

The state visible to the programmer consists of the **program counter** (PC) which contains the address of the next instruction (called RIP on x86-64), the **register file** (which contains heavily used program data), and the **condition codes** which store status information about the most recent arithmetic operation



[11] This chapter mentions however, this course uses x86-64 and AT&T Assembly syntax.

(which is used for conditional branching). The **memory** is byte-addressable and contains code, used data, some OS data and also includes the stack which is used to support procedures.

There are two **data types** in assembly: integers (1, 2, 4, 8 bytes; data values or addresses) and floating point data (4, 8, 10 bytes). As for **code operations** there are three categories: arithmetic functions on register or memory data, data transfer between memory and register, and transfer control (conditional and unconditional branches/jumps).

The **assembler** translates the .s (assembly instructions generated by the compiler) file into .o (object code) which contains binary encodings of each instruction and is, save for the linkages, the executable code. The linkages resolved by the **linker**.

x86 Architecture

To move data, the `movx Source, Dest` instruction is used whereas x is one of {b,w,l,q}[12]. The operands, `Source` and `Dest`, can be any one of the following:

- **Immediate**: constant integer data (prefixed with $), encoded as 1, 2, 4, 8 bytes; e.g. `$0x400`
- **Register**: one of 16 integer registers; note some registers reserved or have special uses for particular instructions; e.g. `%r14d`
- **Memory**: 1, 2, 4,8 consecutive bytes at address given by register; e.g. `%rax`

There are two simple memory addressing modes:

- Normal: (R) → Mem[Reg[R]] where the register `R` specifies a memory address; e.g. `movq (%rcx), %rax`
- Displacement: D(R) → Mem[Reg[R] + D] where the register `R` specifies the start of a memory region and the constant displacement `D` specifies the offset; e.g. `movl 8(%ebp)),%edx`

The most general form, however, is:

Indexed:

D(Rb, Ri, S) → Mem[Reg[Rb] + S · Reg[Ri] + D]

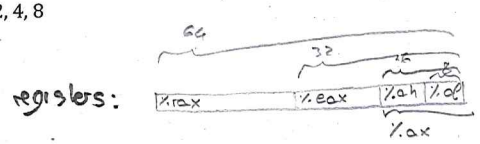
- D: constant displacement of 1, 2, 4 bytes
- Rb: base register: any of 16 integer registers
- Ri: index register: any except for `%rsp`
- S: scale: 1, 2, 4, 8

With the following special cases:

(Rb, Ri) → Mem[Reg[Rb] + Reg[Ri]] (S=1, D=0)
D(Rb, Ri) → Mem[Reg[Rb] + Reg[Ri] + D] (S=1)
(Rb, Ri, S) → Mem[Reg[Rb] + S · Reg[Ri]] (D=0)

Register	Purpose
%rax	Accumulate
%rbx	Base
%rcx	Counter
%rdx	Data
%rsi	Source index
%rdi	Destination index
%rsp	Stack pointer
%rbp	Base pointer
%rip	Instruction pointer
%r8 ... %r15	
%rsr	Status (flags)

Abbr.	Meaning	Bytes
q	Quad word	8
l	Long word	4
w	Word	2
b	Byte	1



[12] See table on the right.

This address computation combined with the `lea`[13] `Src,Dest` instruction can also be (ab-)used to compute addresses without a memory reference and to compute arithmetic expressions of the form $x + k \cdot y, k \in \{1,2,4,8\}$.

x86 integer arithmetic

```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;          leal   (%rdi,%rsi), %eax     # eax = x + y
  int t2 = z+t1;         addl   %edx, %eax            # edx = z + eax
  int t3 = x+4;          leal   (%rsi,%rsi,2), %edx   # edx = y * 3
  int t4 = y * 48;       sall   $4, %edx              # edx *= 16
  int t5 = t3 + t4;      leal   4(%rdi,%rdx), %ecx    # ecx = x + 4 + edx
  int rval = t2 * t5;    imull  %ecx, %eax            # eax *= ecx
  return rval;           ret
}
```

Condition codes

The condition codes are implicitly set (as a "side effect") by the arithmetic operations (but not by `lea`). Or they are set explicitly by compare instructions (`cmpx`[14] `Src2,Src1`) or by test instructions (`testx`[15] `Src2,Src1`) (which is very useful together with bitmasks).

Single bit registers (condition codes)

- `CF` Carry Flag (for unsigned)
- `SF` Sign Flag (for signed)
- `ZF` Zero Flag
- `OF` Overflow Flag (for signed)

The `setx` family of instructions sets a single byte based on combinations of condition codes. The `jx` instructions jump to different parts of the code depending on condition codes.

8 Compiling C Control Flow

if-then-else statements

```
#include <stdio.h>

int putmax(int x, int y)
{
    int result;
    if (x <= y) {
        goto Else;
    }
    result = printf("%d\n", x);
    goto Done;
Else:
    result = printf("%d\n", y);
Done:
    return result;
}
```

```
putmax:
    subq    $8, %rsp          } Setup
    cmpl    %esi, %edi        } Test
    jle     .L2               }
    movl    %edi, %edx        }
    movl    $.LC0, %esi       }
    movl    $1, %edi          } Body 1
    movl    $0, %eax          }
    call    __printf_chk      }
    jmp     .L3               }
.L2:                          }
    movl    %esi, %edx        }
    movl    $.LC0, %esi       }
    movl    $1, %edi          } Body 2 (else)
    movl    $0, %eax          }
    call    __printf_chk      }
.L3:                          }
    addq    $8, %rsp          } Finish
    ret                       }
```

[13] "load effective address"
[14] `cmpl b,a` is like computing $a - b$ without setting the destination
[15] `testl b,a` is like computing `a & b` without setting the destination

An `if-then-else` statement consists of a test, and one (two) `if-then` branch (and one `else` branch). The test is an expression returning an integer whereas anything other than 0 is interpreted as `true` and 0 is interpreted as `false`. Furthermore, any conditional expression can be translated into a `goto` version. This **`goto` version**, which has separate regions for the `then` and `else` expressions (and executes the appropriate one), is the typical translation of a conditional expression into assembly.

```
   nt = !Test;
   if (nt) goto Else;
   val = Then-Expr;
   . . .
   goto Done;
Else:
   val = Else-Expr;
Done:
   return
```

or

```
   nt = !Test;
   if (nt) goto Else;
   val = Then-Expr;
   . . .
Done:
   return
Else:
   val = Else-Expr;
   goto Done;
```

Another way to translate a conditional into assembly is by using a **conditional move**. This makes use of the `cmovC src, dest` instruction which moves a value from `src` to `dest` if the condition C holds. This has the advantage of being more efficient than conditional branching (simpler control flow) but it introduces overhead since both branches are evaluated. Consequently this approach cannot be used if the `then` or `else` expressions have side effects or when they are too expensive.

```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff: # x in %edi, y in %esi
  movl    %edi, %eax
  subl    %esi, %eax
  movl    %esi, %edx
  subl    %edi, %edx
  cmpl    %esi, %edi
  cmovle  %edx, %eax
  ret
```

`do-while` loops

A `do-while` loop uses a backward branch to continue looping. The branch is only taken when the `while` condition holds. As an implication, the loop body is already executed *before* the first check is performed (no matter whether that check returns `true` or `false`). For assmebly translation, the goto-version-method is used again since this allows producing very hardware-like code in C.

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);

  return result;
}
```

```
int fact_goto(int x)
{
  int result = 1;
Loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto Loop;
  return result;
}
```

```
do
  Body
  while (Test);
```

```
Loop:
  Body
  if (Test)
    goto Loop
```

`while` loops

Converting a `while` loop into a `goto` version is very similar to the `do-while` loop's translation save for an extra test before the loop is entered.


```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

```
int fact_while_goto1(int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
Loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto Loop;
done:
  return result;
}
```

```
int fact_while_goto2(int x)
{
  int result = 1;
  goto middle;
Loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto Loop;
  return result;
}
```

There is, however, a new method to perform this translation where the first iteration jumps over the body computation within the loop. This avoids duplicating the code to check the test/condition and unconditional `goto` incur no performance penalty. `for` loops are compiled similarly. The reason for these new "jump-to-middle" loop translations is based on the fact of new(er) processors having almost no overhead when branching unconditionally.

`for` loops

Last but not least, `for` loops are compiled by combining all of the preceding techniques. First, the `for (init;test;update); body;` loop is converted into an `init; while(test); body; update;` loop, which is then translated into a `do-while` version OR a jump-to-middle intermediary, and eventually into a `goto` version.

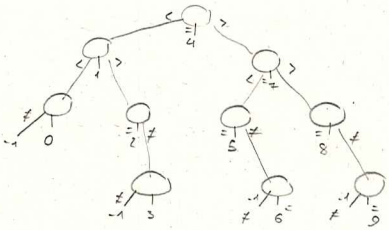
Compact `switch` statements

A compact `switch` statement is a switch where there are e.g. `cases` for a range of numbers plus a `default` case, optionally with multiple `case` labels, missing `cases`, and fall through `cases`. This code block will be converted into a jump table, most likely with `cases` re-arranged to prevent repetitions of the same code (DRY[16]).

Jumps in assembly are either direct where the jump target is denoted by a label (e.g. `.L8`) or indirect, e.g. `jmp *.L4(,%rdi,8)` where the target is loaded from the effective address.

Sparse `switch` statements

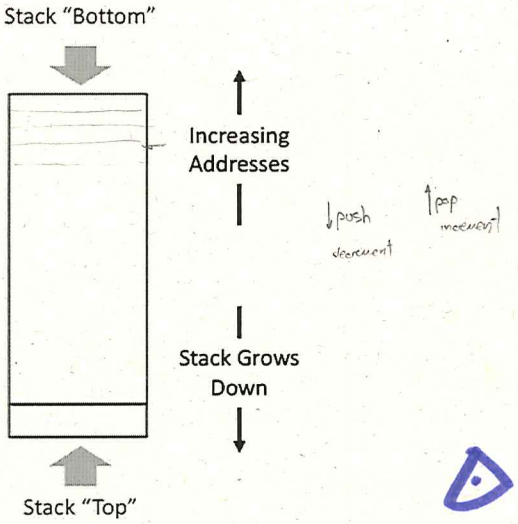
A sparse `switch` statement is impractical to be translated into a jump table and the obvious `if-then-else` doesn't benefit from compiler magic. Such a sparse `switch` can be translated into a binary tree which has logarithmic performance.



[16] "Don't repeat yourself"

Procedure call and return

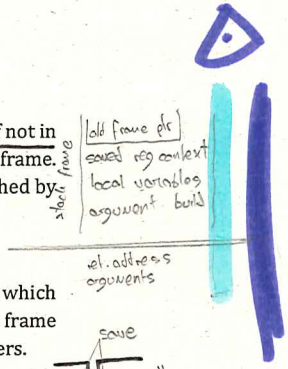
The **stack** is a part of the memory which is managed with stack discipline thus making it a bit different from "normal" memory. The stack grows toward lower address. It is used to save registers when calling procedures, store return values, and pass arguments (if there are more than 6 arguments).



To read and write from and to the stack, push and pop operations are used. These operations increment (pop) or decrement (push) the stack pointer by a number of bytes (using the same syntax like the move instruction) and read from/write to the only argument supplied to the instruction.

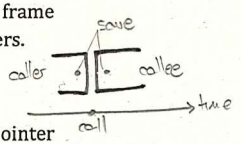
A procedure in Assembly is **called** using `call label`. This pushes the return address onto the stack and jumps to `label`. To **return** simply `ret` is called which pops the address from the stack and jumps to that address.

A *full* stack frame contains (in top-to-bottom order) the argument build, locale variables (if not in registers), the saved register context, and the old frame pointer – all in the current stack frame. And the caller stack frame contains the return address and arguments for this call. It is pushed by the `call` instruction.



x86_64 calling conventions

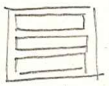
Say procedure `foo()` contains a call to procedure `bar()`. This makes `foo()` the **caller** of `bar()`, which is the **callee**. To ensure data integrity, there are "caller save" (caller saves temporary in its frame before calling) and "callee save"[17] (callee saves temporary in its frame before using) registers.



A few interesting features of the stack frame:

- An entire frame is allocated at once: everything can be accessed relative to the stack pointer `%rsp`. The allocation can be delayed by temporarily using the red zone[18].
- Deallocation is simple: the stack pointer is incremented; no need for base/frame pointer.

⚠ *Slides 65 f. omitted due to non-comprehension of the writer of this document. Any input is appreciated!*



[17] "save", not "safe"!

[18] "In computing, a red zone is a fixed-size area in a function's stack frame beyond the return address which is not preserved by that function. The callee function may use the red zone for storing local variables without the extra overhead of modifying the stack pointer. This region of memory is not to be modified by interrupt/exception/signal handlers. The x86-64 ABI used by System V mandates a 128-byte red zone, which begins directly after the return address and includes the function's arguments." (Wikipedia)

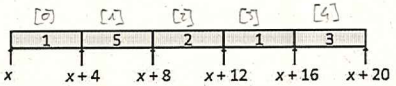
9 Compiling C Data Structures

One-dimensional array

Before we can have a look at how one-dimensional arrays are compiled, we need to have a recap the different amounts of bytes required for each numerical datatype. Integral types are stored and operated on in general integer registers and whethter they are treated as signed or unsigned depends on the instructions used. Floating points numbers are stored and operated on in floating point registers.

	Intel	GAS	Bytes	C
Integral	byte	b	1	char
	word	w	2	short
	double word	l	4	int
	quad word	q	8	long int
Float	single	s	4	float
	double	l	8	double
	extended	t	10/12/16	long double

To allocate an array `A` defined as `T A[L]` of datatype `T` and length `L`, a **contiguous region in memory** of size `L * sizeof(T)` bytes is allocated. As an example, an array `char string[12]` needs 12 bytes whereas `char *p[3]` requires 24 bytes on x86-64.

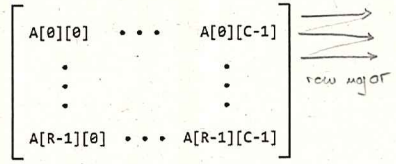


Reference	Type	Value
val[4]	int	3
val	int *	x
val + 1	int *	x + 4
&val[2]	int *	x + 8
val[5]	int	??
*(val+1) = val[1]	int	5
val + i	int *	x + 4 i

Arrays in C-Assembly combination can also be used as pointers or rather elements of arrays can be accessed using pointer arithmetic. Note however, no bound checking is performed, out-of-range behavior is implementation-dependent, and different arrays may not be allocated in the same relative order. When accessing an array in Assembly, a common strategy is to have the starting address in one array and the index in another array, and then use a memory reference with the scale factor set to the size of an array element.

Nested arrays

A definition like `T A[R][C]` is an array of `R`, contiguously allocated elements whereas each element is an array of `C` elements of type `T`, also allocated contiguously. Every element of `T` requires `K` bytes resulting in a total array size of `R*C*K` bytes. This **row-major** ordering of all elements is guaranteed. The starting address of a row `i` is given by `A+i*(C*K)` and its type (of `A[i]`) is simply an array of `C` elements of type `T`. Accessing an element of a nested array in Assembly is similar to one-dimensional arrays, except for an additional row offset multiplication. Pointer arithmetic allows "strange" arrays indices such as -1 to be used and produce a valid and expected result.



⚠ see why not guaranteed

$A[i,j] = A + i*(C*K) + j*K$

Multi-level arrays

Multi-level arrays (also jagged arrays or arrays of arrays) are, as the name implies, an array of arrays. This means, the "first level" of arrays are pointers which point to arrays of some type `T`. This adds a level of indirection and thus requires 2 memory reads: one to get to pointer to the row arrays and the other one to access the element within the arrays.

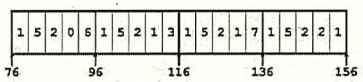
Nested array

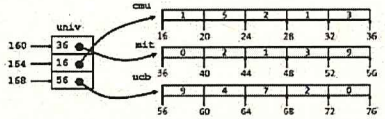
```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

Multi-level array

```
int get_univ_digit
  (int index, int dig)
{
  return univ[index][dig];
}
```

syntax is the same





Access looks similar, but element:

(cast) char * k int

```
Mem[pgh+20*index+4*dig]
```

type int*

2 mem reads, ptr

```
Mem[Mem[univ+8*index]+4*dig]
```

type int**

	Nested arrays	Multi-level array = dynamic nested array
Strengths	- The C compiler handles doubly subscripted arrays - Very efficient code is generated - The multiply in index computation is avoided (?)	- Can create a matrix of any size
Limitations	- Only works for fixed size arrays	- Index computation has to be done explicitly when programming - Accessing a single element is costly - Involves multiplication

Structures

Structures are contiguously allocated regions of memory with members, which are references by names, of possibly different types. The offset of each structure member is determined at compile time (see next section) and thus also the pointer to each element.

Alignment

General rule: if a primitive data type requires ***K* bytes**, the address must be **a *multiple* of *K*** (this varies by architecture and OS)[19]. This allows memory to be access by *aligned* chunks of 4 or 8 bytes, also because it is inefficient to load/store datum[20] which spans quad word boundaries.[21] The compiler inserts **gaps** in a structure to ensure correct alignment of fields.

These alignment rules have to be satisfied within a structure but also overall; each structure has an alignment requirement K (initial address and the length of the structure have to be multiples of K) where K is the alignment of its **largest element**. As a consequence, space can be saved by e.g. putting large datatypes first in the structure's definition.

x86-64/i032 Windows: K=8
i032 Linux: K=4

[19] This implies for a datatype of size 2^n to be aligned with the lowest n bits of the address to be 0. In the case this holds for `char` (1 byte), `short` (2 bytes), `int` and `float` (4 bytes), and double and `char *` (8 bytes), but not for `long double` on Linux (16 bytes, yet aligned to 8-byte boundary).

[20] "datum" is the singular of "data"

[21] Virtual memory is very tricky when a datum spans 2 pages.

accessing:

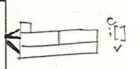
Arrays of structures

In an array of structures, alignment requirements have to be satisfied for every element.

Unions

Unions are allocated according to the largest element and only one field can be used at a time.

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



10 Code Vulnerabilities

Worms and Viruses

A **worm** is a program which can run by itself and can propagate a fully working version of itself to other computers. This in stark contrast to a **virus** which adds itself to other programs and cannot run independently. (to hide)

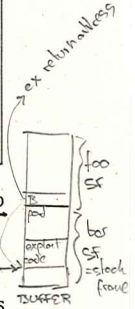
Stack overflow bugs

Consider the Unix implementation of `gets()`. It completely lacks any limit on the number of characters to be read, just like similar functions like `strcpy` or the `scanf` family with `%s`. This weakness combined with a too small buffer can lead to a stack overflow (manifested in a segmentation fault). While a crash is simply annoying, this strategy can also be exploited to overwrite the return address of a function to an address within the buffer and thus making the program jump to the exploit code.

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

end of file

dereference null pointer



understand example
exploit code = executable malicious code

Stopping overrun bugs

Exploits such as the above can be prevented by using library routines which limit string lengths (`fgets` instead of `gets`, `srncpy` instead of `strcpy`). Additonally, there are now system-level protections in place which e.g. **randomize stack** offsets which makes address prediction harder. There are also non-executable code segments. (only read-write)

don't use scanf with %s!
you don't know where you'll be returning

XDR (extern data representation)

Omitted.

11 Memory Allocation

(malloc: critical for performance at every turning)

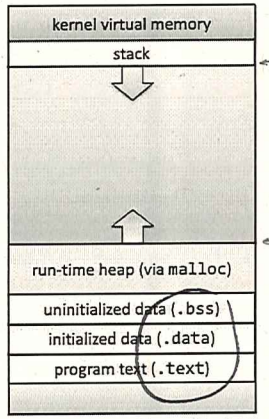
"Sizes of needed data structures may only be known at runtime."

```
#include <stdlib.h>
void *malloc(size_t size)
        Successful: returns a pointer to a memory block of at least
        size bytes (typically) aligned to 8- or 16-byte boundary. If
        size == 0, returns NULL
        Unsuccessful: returns NULL (0) and sets errno
void free(void *p)
        Returns the block pointed at by p to pool of available
        memory. p must come from a previous call to malloc() or
        realloc()
void *realloc(void *p, size_t size)
        Changes size of block p and returns pointer to new block.
        Contents of new block unchanged up to min of old and
```

64b

→ good practice to null pointer afterward's

min(old.size, new.size)



void * calloc (size_t nitems, size_t size)
return ptr to new block of (nitems.size) bytes at least, zeroed out. often aligned.

new size. Old block has been `free()`'d (logically, if `new != old`)

The problem

Assume: memory is word addressed and each word can hold a pointer, which in x86-64 is 64 bits.

A program can issue an arbitrary sequence of `malloc()` and `free()` (only to previously `malloc()`'d blocks) requests. An allocator has no control over the number or size of the allocated blocks yet has to respond immediately to `malloc()` (no reordering/buffering possible) while also aligning the blocks correctly. Of course, only free memory can be manipulated and blocks can't be moved once `malloc()`'d (no compaction possible).

One performance goal is **throughput** i.e. for a given sequence of `malloc` and `free` requests to maximize throughput and maximize peak memory utilization, which are often conflicting. The throughput is the number of completed requests per unit of time.

The other performance goal, **peak memory utilization** requires some level of formalism:

- Given some sequence $R_i, i \in [0, n-1]$ of `malloc` and `free` requests.
- The **aggregate payload** P_k is defined as: `malloc(p)` results in a block with a payload of `p` bytes. After request R_k has completed, the aggregate payload P_k is the sum of currently allocated payloads i.e. all `malloc()`'d stuff minus all `free`'d stuff.
- The **current heap size** H_k: assume H_k is monotonically non-decreasing (it grows when the allocator uses `sbrk()`).
- The **peak memory utilization after k requests**: $U_k = \max_{j<k} P_j / H_k$

Implementation Issues

- How to know how much memory is being `free()`'d when it is given only a pointer (and no length)?
- How to keep track of the free blocks?
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?
- How to pick a block to use for allocation – many might fit?
- How to reinsert a freed block into the heap?

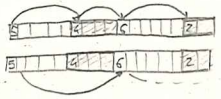
One cause for poor memory utilization is **fragmentation** which can either be **internal** or **external**.

For a given block, internal fragments occurs if the payload is smaller than the block size. This is caused by the overhead of maintain heap data structure, padding for alignment purposes, or explicit policy decisions. Therefore this kind of fragmentation depends only on the pattern of previous requests which makes it easy to measure.

External fragmentation on the other hand depends on the pattern of future requests which makes it difficult to measure. It occurs when there is enough aggregate heap memory but no single free block is large enough.

To know how much to free, the standard method is to keep the length of a block in the word preceding the block (called header field or just header). This, obviously, requires an extra word for every allocated block. To keep track of free lists, the following methods can be used:

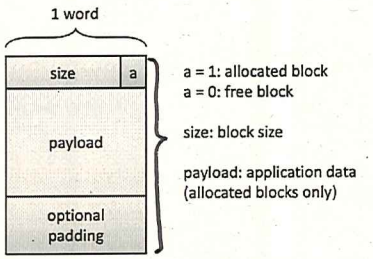
Method 1: Implicit list using length – links all blocks
Method 2: Explicit list among the free blocks using pointers
Method 3: Segregated free list (different free lists for different size classes)



Method 4: Blocks sorted by size (this can use a balanced tree (e.g. red-black) with pointers within each free block and the length used as a key)

Implicit free lists

For an implicit free list to work, 2 pieces of information per block need to be stored: its length and whether it's allocated. To save one word and only use a single word to store this information, the following trick is used: if the blocks are aligned, some low-order address bits are always 0, making it perfect to be used as an allocated/free flag; it only has to be masked out when reading the word.[22]



For exercise purposes, the following encoding is often used: `length/is_allocated`. The end of the list is marked by `0/1`.

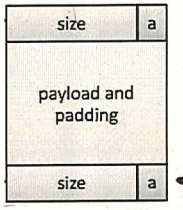
Strategies to find a free block	First fit	Next fit	Best fit[23]
	Search the list from the beginning and choose the first block which fits. While it works, it can take linear time in the amount of total blocks. It can also cause splinters at the beginning of the list.	Similar to first fit but the search is continued where the previous one left off. This should be faster than first-fit since unhelpful blocks aren't rescanned yet fragmentation might be worse.	The whole list is searched for the "best" block where "best" refers to a block which fits with the fewest bytes left over. This keeps fragments small (less fragmentation) but is slower than first fit.

When a free block is found it can either be allocated in full or split. **Splitting** makes sense when allocated space is smaller than free space.

To free a block, it suffices to clear the allocated flag yet this could lead to so called "false fragmentation" which results in having free space which isn't found by the allocator.

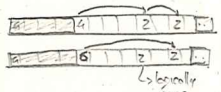
Coalescing

Coalescing is the process of joining a block with the next/previous block given it is free. This is done to have larger free blocks. To perform bidirectional coalescing, **boundary tags** are used. They replicate the size/allocated information at the bottom/end of free blocks which allows the list to be traversed backwards (but requires extra space and leads to internal fragmentation).



Coalescing is either immediate (after `free()`) or deferred (e.g. when the list is scanned for `malloc()` or at an external fragmentation threshold).

policies: (decisions to be made)
- placement (first/next/best fit)
- splitting (when do we split free block?)
- coalescing (imm./deferred)

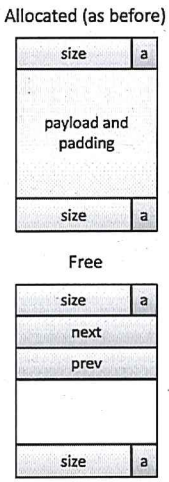


[22] `*p & -2`

[23] This is approximated by segregated free lists without having to search the entire free list.

Explicit free lists

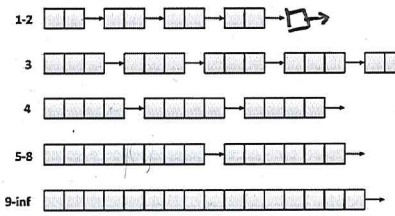
Instead of maintaining a list of all blocks, only **free** blocks are tracked. And since the next free block could be anywhere, not only sizes but also forward and backward pointers need to be stored but since only free blocks are tracked, the payload area can be used for the pointers. Boundary tags are still necessary for coalescing.



A newly freed block can be inserted into the list using different policies.

Policy	LIFO	Address-ordered
	Insert freed block at the beginning of the free list	Insert freed blocks so that free list blocks are always in address order: addr(prev) < addr(curr) < addr(next)
Pro	Simple and constant time	Requires search
Con	Studies suggest fragmentation is worse than address ordered	Studies suggest fragmentation is lower than LIFO

Segregated free lists



For every **size class** of blocks there exists a separate free list. Often small blocks have one list for each size whereas larger sizes are grouped by powers of two.

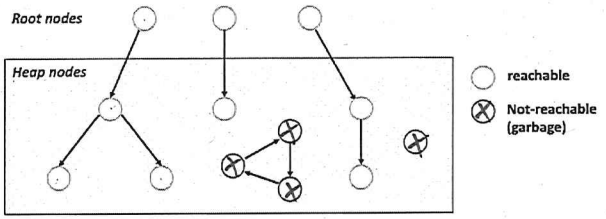
To **allocate** a block of size n the appropriate free list is searched for a block of size $m > n$. If a block is found, the block is split and the fragment is (optionally) placed on the appropriate list. If no block is found, the next larger class will be tried. If, even after repeating this process, no block is found, additional heap memory from the OS is requested and allocate a block of n bytes in this new memory, placing the remainder as a single block in the largest size class. When **freeing** a block, the memory is coalesced and (optionally) placed on the appropriate list.

Advantages of seglists are higher throughput (logarithmic time for power-of-two classes) and better memory utilization because the first-fit search of a segregated list approximated a best-fit search of the entire heap; in the extreme case where every block has its own size class, it is equivalent to best-fit.

Garbage collection

Garbage collection is the process of automatically (implies: the application doesn't have to free the memory by itself) reclaiming heap-allocated storage. While it is not possible to predict the future i.e. know what is going to be used depends on conditionals, if a block doesn't have any pointers to it, it can be assumed it will not be used anymore. This requires certain assumptions about pointers. First of all, the memory manager needs to be able to distinguish between pointers and non-pointers. Secondly, all pointers have to point to the start of the block. And lastly, pointers cannot be hidden (e.g. by coercing them to an `int`).

Let the memory be a graph with each block being node and each pointer being an edge. Furthermore locations in the heap which contain pointers into the heap are called root nodes (e.g. registers, locations on the stack, global variables). A node is said to be reachable if there is a path from any root to that node. Otherwise it's garbage.



To implement a **Mark and Sweep** garbage collector you can build on top of the `malloc/free` packge. `malloc` is called until it runs of out space. When that happens, the extra **mark bit** in the head of each block is used in a two-step process.

1. **Mark**: start at the roots and set mark bit on each reachable block
2. **Sweep**: scan all blocks and free blocks that are not marked

For a simple implementation, the following is assumed:

- `new(n)`: returns pointer to a new block with all locations cleared
- `read(b,i)`: read location `i` of block `b` into register
- `write(b,i,v)`: write `v` into location `i` of block `b`

Each block will have a header word which is addressed as `b[-1]` for block `b` (different uses in different collectors).

- `is_ptr(p)`: determines whether `p` is a pointer
- `length(b)`: returns the length of block `b`, not including the header
- `get_roots()`: returns all the roots

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
    if (!is_ptr(p)) return;
    if (markBitSet(p)) return;
    setMarkBit(p);
    for (i=0; i < length(p); i++) {
        mark(p[i]);
    }
    return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
    while (p < end) {
        if markBitSet(p) {
            clearMarkBit();
        } else if (allocateBitSet(p)) {
            free(p);
        }
        p += length(p);
    }
}
```

A conservative implementation of the mark & sweep algorithm in C uses `is_ptr()` to determine whether a word is a pointer by checking if it points to an allocated block of memory. However, in C pointers can point to the middle of a block. To solve this problem (i.e. to find the beginning of a block) a balanced binary tree is used to keept rack of all allocated blocks (the key is the start-of-block). Balanced-tree pointers can be stored in the header.

Memory pitfalls

Dereferencing bad pointers	`// The classic scanf bug`

	`int val;`
	`...`
	`scanf("%d", val);` → newline when pressing ↵ is counted as char
	scanf("%f", &number1);
	getchar();

Reading uninitialized memory	`// Assuming that heap data is initialized to zero`
	`/* return y = Ax */`
	`int *matvec(int **A, int *x) {`
	`  int *y = malloc(N*sizeof(int));`
	`  int i, j;`
	`  for (i=0; i<N; i++) {`
	`  for (j=0; j<N; j++) {`
	`  y[i] += A[i][j]*x[j];`
	`  }`
	`  } → doesn't start out as 0!`
	`  return y;`
	`}`

Overwriting memory	`// Allocating the (possibly) wrong sized object`
	`int **p;`
	`p = malloc(N*sizeof(int));` → should be int*
	`for (i=0; i<N; i++) {`
	`  p[i] = malloc(M*sizeof(int));`
	`}`
	
	`// Off-by-one error`
	`int **p;` → should be <
	`p = malloc(N*sizeof(int *));`
	`for (i=0; i<=N; i++) {`
	`  p[i] = malloc(M*sizeof(int));`
	`}`
	
	`// Not checking the max string size`
	`char s[8];`
	`int i;`
	`gets(s); /* reads "123456789" from stdin */` → too long
	buffers attacks →
	`// Misunderstanding pointer arithmetic`
	`int *search(int *p, int val) {`
	`  while (*p && *p != val) {` → scan everything until found
	`  p += sizeof(int);`
	`  } → should be 1`
	`  return p;`
	`}`
	
	`// Referencing a pointer instead of object it points to`
	`int *BinheapDelete(int **binheap, int *size) {`
	`  int *packet;`
	`  packet = binheap[0];  //ptr`
	`  binheap[0] = binheap[*size - 1];`
	`  *size--;`
	`  Heapify(binheap, *size, 0);`
	`  return(packet);`
	`}` → should be *packet
	heap of pointers

Referencing nonexistent variables	`// Forgetting that local variables disappear when a function returns`
	`int *foo () {`
	`  int val;`
	`  return &val;  // disappears`
	`}` (but you can return value)

Freeing blocks multiple times	`// nasty!`
	`x = malloc(N*sizeof(int));`
	`  <manipulate x>`
	`free(x);`
	`y = malloc(M*sizeof(int));`
	`  <manipulate y>`
	`free(x);  // danger!`

Referencing freed blocks	`// evil`
	`x = malloc(N*sizeof(int));`
	`  <manipulate x>`

	`free(x);`
	`...`
	`y = malloc(M*sizeof(int));`
	`for (i=0; i<M; i++) {`
	`  y[i] = x[i]++;` //it has been freed (wouldn't know what one might find)
	`}`

Failing to free blocks	`// slow, long-term killer`
	`foo() {`
	`  int *x = malloc(N*sizeof(int));  //stays in memory`
	`  ...`
	`  return;`
	`}`

Memory leaks	`// Freeing only part of a data structure`
	`struct list {`
	`  int val;`
	`  struct list *next;`
	`};`
	`foo() {`
	`  struct list *head = malloc(sizeof(struct list));`
	`  head->val = 0;`
	`  head->next = NULL;`
	`  <create and manipulate the rest of the list>`
	`  ...`
	`  free(head);  // rest of list is left in memory`
	`  return;`
	`}`

→ good to find bad ptr dereferences

To find memory bugs, conventional debuggers such as `gdb` can be used or it can be done by debugging `malloc` (e.g. UTorronto CSRO `malloc`) which wraps around `malloc` and performs boundary checking. There are `malloc` implementations which contain checking code. Further tools include binary translators (e.g. valgrind, Purify) and garbage collection (e.g. Boehm-Weiser Conservative GC).

on malloc and free

→ can check each individual ref at runtime
- bad ptr
- overwriting
- referencing outside allocated blocks

12 Linking

Programs are translated and linked using a **compiler driver**. This process is called **static linking**. This compiler driver (e.g. `gcc`) first generates **object files** (`.o`) from source files (`.c`) by using translators (`cpp,cc1,as`). These, separately compiled and relocatable, object files are then linked using a linker (`ld`) into a fully executable object file. This file contains code and data for all functions defined in the source file.

→ intermediate step: .s (assembly)

The advantage of linkers is **modularity**: instead of writing one huge program, it can be written as many small source files. It also enables the programmer to use common functions from libraries. Another advantage is **efficiency**: it is much faster to re-compile one file after changing it than the whole program. And by using libraries a lot of space can be saved. This is done by aggregating common functions into a single file which makes the executables only have code for functions they actually use (both in file and in memory).

Linkers work in a two-step process:

ref → def

counter (int) 4bytes @ 0x...

Step 1: **Symbol resolution**: a program defines and references symbols (variables, functions). These definitions are stored in a symbol table by the compiler. This table is an array of structs where each entry includes name, type, size, and location. The linker then associates each symbol reference with exactly one symbol definition.

Step 2: **Relocation**: during the relocation phase, separate code and data sections are merged into a single section. The symbols are relocated from the relative locations in the

.o files to their final and absolute memory locations in the executable. This of course requires an update on all symbol references.

Object files

Relocatable object file (.o)	Executable object file	Shared object file (.so/dll)
Contains code and data in a form that can be combined with other relocatable object files to form executable object file. Each .o file is produced from exactly one source (.c) file	Contains code and data in a form that can be copied directly into memory and then executed. (to execute, first load into mem instructions & variables)	Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.

ELF object file format

ELF header	Word size, byte ordering, file type (.o, exec, .so), machine type etc.
Segment header table	Page size, virtual address memory segments (sections), segment sizes
.text	Code (program instructions)
.rodata	Read only data: jump tables ...
.data	Initialized global variables
.bss	Uninitialized global variables, "block started by symbol"/"better save space", has section header but occupies no space
.symtab	Symbol table, procedure and static variables names, section names and locations
.rel.txt	Relocation info for .text section, address of instructions that will need to be modified in the merged executable
.rel.data	Relocation info for .data section, addresses of pointer data that will need to be modified in the merged executable
.debug	Info for symbolic debugging (gcc -g)
Section header table	Offsets and sizes for each section

.init?

Linker symbols

Global symbols	External symbols	Local symbols
Symbols defined by module *m* that can be referenced by other modules. E.g.: non-static C functions and non-static global variables.	Global symbols that are referenced by module *m* but defined by some other module.	Symbols that are defined and referenced exclusively by module *m*. E.g.: C functions and variables defined with the static attribute. Local linker symbols are **not** local program variables

Additionally, these symbols are either strong or weak. **Strong** symbols are procedures and initialized globals whereas **weak** symbols are uninitialized globals.

strong = initialized
weak = uninitialized

The linker works through the symbols following three rules:

Rule 1: Multiple strong symbols are not allowed. Each item can be defined only once.
Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol. References to the weak symbol resolve to the strong symbol.

Rule 3: If there are multiple weak symbols, pick an arbitrary one.[24]

Global variables should be avoided if possible. Instead static could be used or the global variable should be initialized. If an external global variable is used, extern should be used.

*see linker puzzles (p22)

Static libraries

To package functions which are commonly used (e.g. math, I/O, memory management, string manipulation) by programmers, statics libraries (.a, for archive files) are used ideally. They concatenate related relocatable object files into a single file with an index (aka archive). The linker now also looks in these archives when resolving unresolved external references. Should an archive file member resolve the reference, it is linked into the executable. When creating[25] such an archive, the archiver allows for incremental updates.

To use static libraries, the linker's algorithm works as follows:

es libc.a //c standard
libm.a // math lib

- Scan .o and .a files in the command line order
- During the scan, keep a list of the current unresolved references
- As each new .o or .a file, obj, is encountered, try to resolve each unresolved reference in the list against the symbols defined in obj.

One of the problems is the command line *order* being relevant. This can be "solved" by putting libraries at the end of the command line.

Shared libraries

Static libraries have a few disadvantages: there is some level of duplication in the stored executables (e.g. due to libc) which leads to duplication in the running executables. And since they are loaded/linked statically, a minor bug fix in a system library requires each application to explicitly relink. The solution is to use shared libraries (aka dynamic link libraries, DLL, .so). The object files containing code and data are linked into the application *dynamically*, either at *load-time* or at *run-time*.

Using **load-time linking** the linking happens when the executable is first loaded. This is the common case for Linux and handled automatically by the dynamic linker (e.g. for libc.so).

When using **run-time liming** the dynamic linking occurs after the program has begin. This is done by the dlopen() interface and is commonly used in high-performance web servers and for runtime library interpositioning.

Shared library routines can be shared by multiple process (cf. shared pages).

a historical curiosity – omitted

13 Floating Point

(not used that much in syst. and databases)

Representing floating-point numbers

fixed point

Using fractional binary numbers ($\sum_{k=-j}^{i} b_k \cdot 2^k$) allows for easy divide and multiply operations (using shifts) but they can only exactly represent numbers of the form $x/2^k$. That's why a standard was desperately needed 30 years ago and IEEE 754 was born.

[24] This can be overridden with gcc -fno-common
[25] ar rs libc.a atoi.o printf.o … random.o

Numerical form	Encoding
$(-1)^s M 2^E$ with sign bit s, significand (or mantissa) M (normally a fractional value $\in [1.0,2.0)$) and exponent E which values it by a power of two	The MSB is the sign bit s, followed by `exp` which *encodes* E, and `frac` which *encodes* M.

Types of IEEE floating-point numbers

Different types of IEEE floating point numbers offer different precisions:

Name	s	exp	frac	Total bits
IEEE 754 Single Precision	1	8	23	32
IEEE 754 Double Precision	1	11	52	64
Intel Extended Precision	1	15	63	80
IEEE 754 Quadruple Precision	1	15	112	128

In C there are only two guaranteed levels, `float` and `double`. The type `long double` can be any non-single-precision type. Casting between `int`, `float`, and `double` changes the bit representation. Converting to `int` truncates the fractional part ("round towards zero") and is undefined when out of range/NaN. Converting `int` to `double` is exact provided the `int` has less or equal to 53 bits word size. Converting to a `float` rounds according to rounding mode.

	Normalized values	Denormalized values	Special values
Condition	`exp != 000…0 && exp != 111…1`	`exp == 000…0`	`exp == 111…1`
	Exponent is coded as a biased value: - $E = Exp - Bias$ - Exp unsinged value of exp - $Bias = 2^{e-1} - 1$ where e is the number of exponent bits (see table below) Mantissa is coded with implied leading 1: $M = 1.xxx\ldots x_2$ - xxx…x are the bits of frac - Minimal when 000…0 ($M = 1.0$) - Maximal when 111…1 ($M = 2.0 - \varepsilon$)	Exponent value: $E = -Bias + 1$ (instead of $E = 0 - Bias$) Significand coded with implied leading 0, $M = 0.xxx\ldots x_2$ Cases: - `frac == 000…0`: value 0 (actually ± 0, both exist) - `frac != 000…0`: numbers very close to 0.0, lose precision as they get smaller; equispaced	Cases: - `frac == 000…0`: represents ∞ for overflowing operations; negative and positive exist - `frac != 000…0`: Not-A-Number (NaN); for when no numeric value can be determined

Precision	Exponent bias	Exp range	E range
Single	127	1…254	-126…127
Double	1023	1…2046	-1022…1023

Example

Value: `float F = 15213.0;`

$15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13}$

Significand: $M = 1.1101101101101_2$, $frac = 1101101101101\,0000000000_2$

Exponent: $E = 13$, $Bias = 127$, $Exp = 140 = 10001100_2$

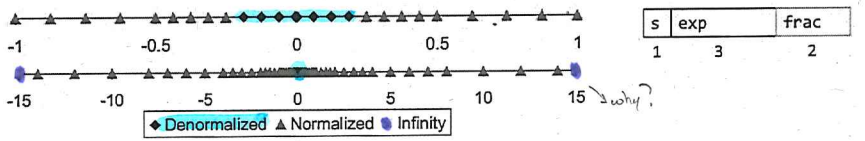
s	exp	frac
0	10001100	11011011011010000000000

Floating point comparison is very similar to unsigned integer comparison save for a few details: Sign bits have to be compared first, +0 and -0 have to be taken into account, and NaNs are problematic since they are greater than any other value.

Floating-point ranges

This section is about a small example from the slides. The first image is about the first format, the second two images about the second format (pictured on the right).

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512 = 1/64	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	



s	exp	frac
1	3	2

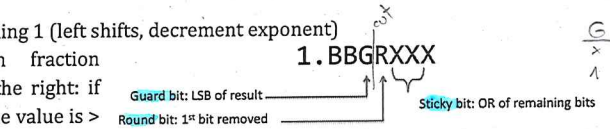
Floating-point rounding

When performing floating point operations which require rounding, the exact result is computed first, and then it is fit into desired precision (which might overflow if the exponent is too large).

IEEE FP uses **round-to-even** by default (no statistical bias) which when exactly hallway between two possible values rounds so that the least significant digit is even.

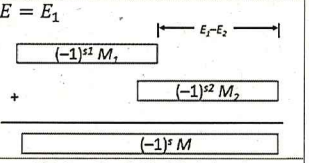
Creating a floating point number is a three-step process:

1. **Normalize** to have a leading 1 (left shifts, decrement exponent)
2. **Round** to fit within fraction
   Consider the image on the right: if `round` and `sticky` are 1, the value is > 0.5 and rounded up. If `guard` and `round` are 1 but `sticky` is 0, it rounds to even.
3. **Postnormalize** to deal with effects of rounding (right shifts, increment exponent)



Floating-point addition and multiplication

	Multiplication	Addition

Problem statement	$(-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}$ $= (-1)^s M 2^E$	$(-1)^{s_1} M_1 2^{E_1} + (-1)^{s_2} M_2 2^{E_2}$ $= (-1)^s M 2^E$
Exact result	$s = s_1 \hat{} s_2$ $M = M_1 \cdot M_2$ $E = E_1 + E_2$	$E = E_1$ 
Fixing	- If $M \geq 2$, shift M right, increment E - If E out of range, overflow - Round M to fit frac precision	- If $M \geq 2$, shift M right, increment E - If $M < 1$, shift M left k positions, decrement E by k - Overflow if E out of range - Round M to fit frac precision
Monotonicty (except for ∞, NaN)	$a \geq b \Rightarrow a + c \geq b + c$	$a \geq b \wedge c \geq 0 \Rightarrow a \cdot c \geq b \cdot c$

Floating-point puzzles

Omitted. ← see

SSE floating point

Omitted.

14 Optimizing Compilers

The compiler is your (shy) friend! – Prof. Roscoe

Where compilers are good and where they run into limitations:

Strong suits	Pitfalls	Limitations
Mapping program to machine: - register allocation, scheduling - dead code elimination - minor inefficiencies elimination	Improving asymptotic efficiency: - programmer has to select best overall algorithm Overcoming optimization blockers: - memory aliasing - procedure side-effects	*If in doubt, the compiler is conservative* Fundamental constraints: - must not change program behavior under any condition Behavior obvious to programmer may be obfuscated by languages and coding styles Analysis is performed only within procedures and based on *static* information

Removing unnecessary procedure calls

Procedure calls and bound checking can be expensive and abstract data types can lead to inefficiencies. If possible and reasonable they should be considered to be removed/inlined. Additionally, **watch your innermost loop.**

Code motion and precomputation

Sometimes the same result gets computed over and over again. This frequency may be reduced by moving code out of a loop (provided the result stays the same). The compiler *might* do this for you.

Strength reduction

A costly operation (e.g. multiplication) can be replaced by a simpler one (e.g. shift). The usefulness of this optimization depends on the machine.

Common subexpressions

Compilers are very bad at exploiting arithmetic properties which e.g. for grid-based coordinate/index calculations (matrix-like e.g. image) are heavily used. In such as a scenario it makes sense to explicitly define and calculate common subexpressions.

Optimization blocker: procedure calls

When the compiler cannot be absolutely sure a procedure will always produce the same result in a loop, it will not inline it and thus call it on every loop iteration. Provided the result doesn't change (as regarded by the programmer) this procedure call can and should be moved outside of the loop.

Compiler usually treats procedure call as a black box that cannot be analyzed.

Optimization blocker: memory aliasing

Two different memory references can write to the same location which can easily be achieved in C. This can be solved by replacing scalars in the innermost loop and copying memory variables which are reused into local variables.

Solution: scalar replacement

Blocking and unrolling

Omitted, see Unrolling, reassociation, multiple accumulators: the code *on page 31 (on page 31)*

15 Architecture and Optimization

A (brief) recap of sequential processor design

A (brief) recap of pipelined processor design

Omitted.[26]

Superscalar processor design

Definition: A superscalar processor can issue and execute multiple instructions in one cycle. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically. Benefit: without programming effort, a superscalar processor can take advantage of the instruction level parallelism that most programs have.

Superscalar performance

While some instructions take more than one cycle, they can be pipelined (this is also a major problem for fast execution: the pipelines have to be kept full). Hard bounds on cycles needed are given by how many cycle an operation itself takes.

Micro-ops and dataflow

When executing a program, the instructions bytes are fetch from memory and the hardware dynamically guesses branches taken/not taken. Instructions are translated into **micro-operations** (for CISC CPUs) which are basically RISC operations for every primitive step performed in the instruction. An instructions typically requires 1-3 micro-ops. Register references are converted

[26] See also: http://studysheets.ch/sheets/digitaltechnik/download

into tags as means of abstraction for combining with other operations. The **goal** is to have each operation utilize a single functional unit.

Dataflow view of instruction execution: view each write as creating new instance of value, operations can be performed as soon as operands available, and no need to execute in original sequence. (op reordering)

Reassociation

Omitted, see Unrolling, reassociation, multiple accumulators: the code *on page 31 (below).*

Combining multiple accumulators and unrolling

Omitted, see Unrolling, reassociation, multiple accumulators: the code *on page 31 (below).*

Branch prediction

The **Instruction Control Unit** must work well ahead of Execution Unit to generate enough operations to keep the EU busy. When the ICU encounters conditional branch, it cannot reliably determine where to continue fetching. A solutions is to guess the branch and begin executing instructions at predicted position but not to actually modify any register or memory data. The resulting branch misprediction penalty is a cost of multiple cycles on a modern processor and can limit performance a lot. (flush)

Unrolling, reassociation, multiple accumulators: the code

Description	Code

Original code	`void combine4(vec_ptr v, data_t *dest)`
	`{`
	`int i;`
	`int length = vec_length(v);`
	`data_t *d = get_vec_start(v);`
	`data_t t = IDENT;`
	`for (i = 0; i < length; i++)`
	`  t = t OP d[i];`
	`*dest = t;`
	`}`

Loop unrolling	`void unroll2a_combine(vec_ptr v, data_t *dest)`
	`{`
	`int length = vec_length(v);`
	`int limit = length-1;`
	`data_t *d = get_vec_start(v);`
	`data_t x = IDENT;`
	`int i;`
	`/* Combine 2 elements at a time */`
	`for (i = 0; i < limit; i+=2) {`
	`  x = (x OP d[i]) OP d[i+1];`
	`}`
	`/* Finish any remaining elements */`
	`for (; i < length; i++) {`
	`  x = x OP d[i];`
	`}`
	`*dest = x;`
	`}`

Loop unrolling with reassociation	`void unroll2ra_combine(vec_ptr v, data_t *dest)`
	`{`
	`int length = vec_length(v);`
	`int limit = length-1;`
	`data_t *d = get_vec_start(v);`
	`data_t x = IDENT;`
	`int i;`
	`/* Combine 2 elements at a time */`
	`for (i = 0; i < limit; i+=2) {`

	`  x = x OP (d[i] OP d[i+1]);`
	`}`
	`/* Finish any remaining elements */`
	`for (; i < length; i++) {`
	`  x = x OP d[i];`
	`}`
	`*dest = x;`
	`}`

Loop unrolling with separate accumulators	`void unroll2sa_combine(vec_ptr v, data_t *dest)`
	`{`
	`int length = vec_length(v);`
	`int limit = length-1;`
	`data_t *d = get_vec_start(v);`
	`data_t x0 = IDENT;`
	`data_t x1 = IDENT;`
	`int i;`
	`/* Combine 2 elements at a time */`
	`for (i = 0; i < limit; i+=2) {`
	`  x0 = x0 OP d[i];`
	`  x1 = x1 OP d[i+1];`
	`}`
	`/* Finish any remaining elements */`
	`for (; i < length; i++) {`
	`  x0 = x0 OP d[i];`
	`}`
	`*dest = x0 OP x1;`
	`}`

unrolling & accumulating

16 Caches

Definition: Computer memory with short access time used for the storage of frequently or recently used instructions or data.

Hits and misses

Say you want to retrieve data in block b. If said block is in the cache, it is called a **hit**, and a **miss** otherwise. When a miss occurs, block b is fetched from memory and stored in cache according to **placement** (where) **and replacement** (victim of eviction) **policies**. There are some metrics which can be used to measure cache performance:

- **Miss rate**[27]: the fraction of memory references not found in cache; typically 3%-10% for L1, <1% for L2
  (misses/accesses) = 1 − hit rate
- **Hit time:** the time to deliver a line in the cache to the processor (incl. time to determine whether line is in the cache); typically 1-2 cycles for L1, 5-20 for L2
- **Miss penalty:** additional time required because of a miss; typically 50-200 cycles for main memory (trend: increasing)

There are, of course, different types of cache misses:

- **Cold (compulsory):** occurs on first access to a block
- **Conflict:** most caches limit blocks to a small subset of the available cache slots (e.g. modulo the address). Thus conflict misses occur when there is enough space but multiple data maps to the same slot
- **Capacity:** active cache blocks (working set) is large than the cache

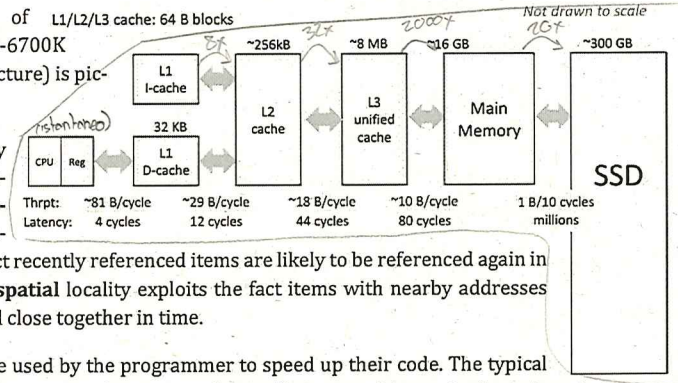
[27] Reason why miss and not hit rate is used: when the miss penalty is large (e.g. 100 cycles for a cache hit time of 1 cycle), 99% hits is double as good as 97%.

- **Coherency:** (see later in this chapter)

The memory hierarchy

Some slides which I consider to be obvious for a 3rd semester computer science student are omitted.

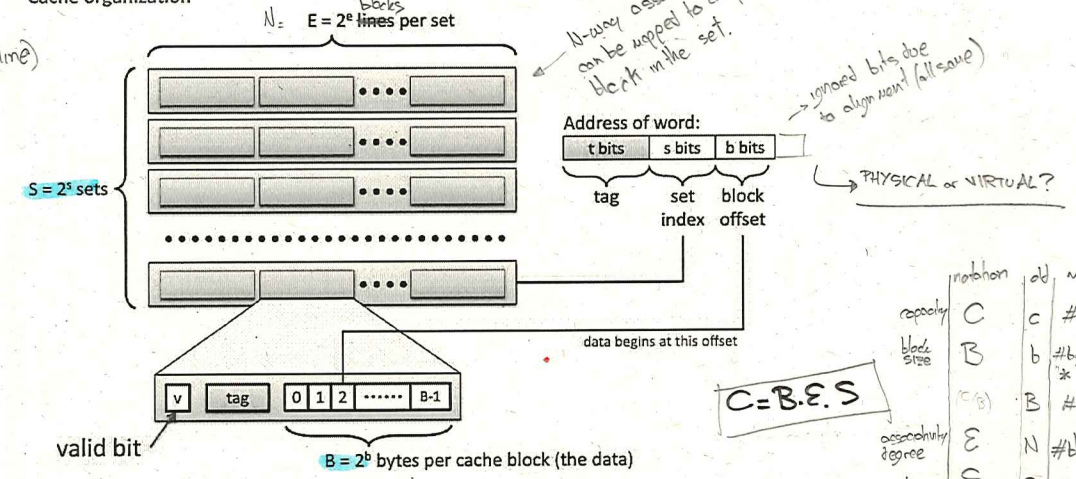
Memory hierarchy of the Intel® Core™ i7-6700K CPU (Skylake architecture) is pictured on the right.



The reason why caches work is because they exploit **locality**: **temporal** locality refers to the fact recently referenced items are likely to be referenced again in the near future and **spatial** locality exploits the fact items with nearby addresses tend to be referenced close together in time.

Spatial locality can be used by the programmer to speed up their code. The typical example of good vs. bad locality (assuming a "typical" computer) is two for loops iterating over a matrix in either row-major (good) or column-major (bad) order.

Cache organization



Cache reads

To read data from the cache, a sequence of operations is performed. First the set is located, then it checks whether any line in the set has a matching tag. If so, it is a hit and data starting at the offset is located. If there is no match, what happens next depends on the type of cache:

- **Direct mapped:** has one line per set ($E = 1$) and simply evicts the old line and replaces it with the new data
- **2-way set-associative:** has two lines per set ($E = 2$) and selects one line the set for eviction and replacement. Selecting the line is according to the replacement policy (random, LRU, ...)

Cache writes

Write-hit	Write-miss
Write-through - Write immediately to memory - ⊕ Memory is always consistent with the cache copy - ⊖ Slow: what if the same value (or line!) is written several times	**No-write-allocate** (writes immediately to memory) - Simpler to implement - Slower code (bad if value subsequently re-read) - Seen with write-through caches
Write-back (much faster) - Defer write to memory until replacement of line - Need a dirty bit (indicates line is different from memory) - ⊕ Higher performance (but more complex)	**Write-allocate** (load into cache, update line in cache) - Good if more writes to the location follow - More complex to implement - May evict an existing value - Common with write-back caches

Software caches (e.g. file system buffer, web browser cache) are much more flexible, often fully associative (using index structures like hash tables), are not necessarily constrained to block transfers, but often have complex replacement policies (also because misses can be very expensive).

Cache optimizations

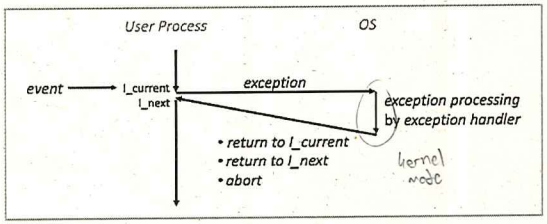
To optimize the use of the cache, code should make use of spatial (access data contiguously) and temporal (access to the same data should not be too far apart in time) locality. This can be achieved by the proper choice of algorithm and loop transformations. The register space is much smaller [than the cache] and requires scalar replacement to exploit temporal locality. Register level optimizations include exhibiting instruction level parallelism (which conflicts with locality).

Blocking (technique to simplify)

Omitted.

17 Exceptions

To change the daily stone-grinding of a processor (i.e. read and execute a sequence of instructions, one after the other, also called **control flow**), jumps/branches and calls/returns can be used. To make the control flow exceptional, exceptions can be used. They react to change in the system state such as data arriving from drive or network, division-by-zero, Ctrl-C input, and the system timer. Exceptions exist at all levels of a computer system – low-lever (hardware + OS) and higher level (context switch, signals, non-local jumps, and language-level exceptions).



Exception vectors and kernel mode

Each type of event has a unique exception number k, whereas k is an index into the exception table (aka interrupt vector). Handler k is called each time exception k occurs. When an exception occurs, the system enters kernel[28] mode.

Synchronous exceptions

Synchronous exceptions are caused by events that occur as a result of executing an instruction. They can be grouped into the following three categories:

Traps	Faults	Aborts
- Intentional - E.g. syscalls, breakpoints, special instructions, opening a file - Control is returned to next instruction	- Unintentional but possibly recoverable - E.g. page faults, protection faults, FP exceptions, invalid memory reference (→ segfault) - Either re-execute current (faulting) instruction or abort	- Unintentional and unrecoverable - E.g. parity error, machine check - Aborts current program

Asynchronous exceptions

Asynchronous exceptions, also called **interrupts**, are caused by events external to the processor and are indicated by setting the processor's interrupt pin. Examples include **I/O** such as Ctrl-C, an arriving network packet or data being ready from disk, and **hard** and **soft reset** interrupts. The handler returns to the next instruction.

Interrupts (whose mechanism is also used for exceptions) work as follows:

1. The CPU interrupt-request line is triggered by an I/O device (edge or level-triggered)
2. The interrupt handler receives the interrupt
3. The interrupt might be **maskable** which allows it to be ignored or delayed
4. The interrupt vector is used to dispatch the interrupt to the correct handler (based on priority)

Rest of this sub-chapter is omitted.

Interrupt controllers

Programmable interrupt controllers solve the problem of e.g. interrupt conflicts and simultaneous interrupts. They map (mapping picked by the OS) physical interrupt **pins** to interrupt **vectors**, buffer **simultaneous** interrupts (deliver each vector separately and make sure not to lose some device's interrupt), **prioritize** interrupts (some devices may interrupt other devices; selected by the OS), and selectively **mask** any individual device's interrupts (useful for high-interrupt rate devices and at boot). Modern PICs also provide inter-processor interrupts, a programmable timer, sophisticated interrupt scheduling etc.

[28] The kernel is the part of the OS which runs in kernel mode. Kernel mode means there's access to system state, some different instructions/registers, different MMU behavior, some exceptions are disabled etc. A kernel *always* is a set of trap handling functions and creates the user-space processes illusion. See also http://studysheets.ch/sheets/operating-systems/download

18 Virtual Memory

Since this was already discussed in the previously mentioned "Operating Systems and Networks" course, I might be a little short on detailed explanations.

The problems with physical memory

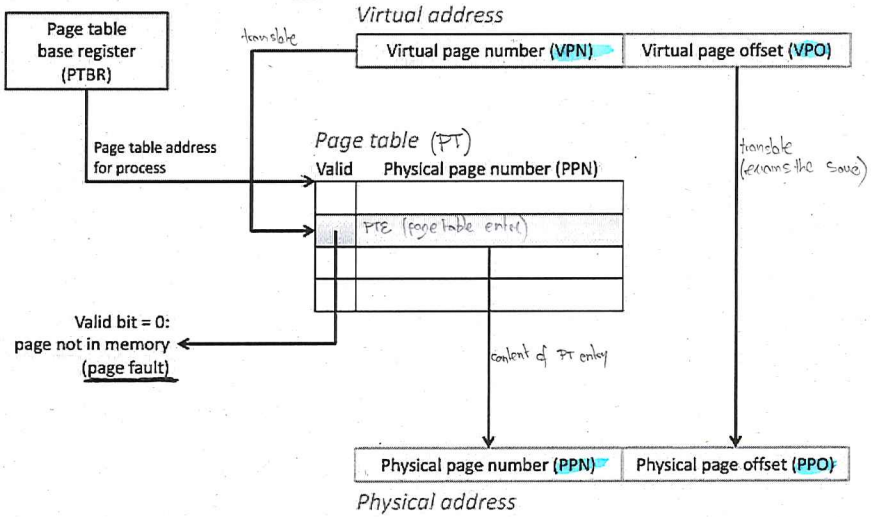
1. 64-bit addresses result in 16 EB memory, but physical main memory is often just a few GB
2. Memory management: what goes where?
3. Protect memory from other processes
4. Share memory with other processes

Solution: address translation

Each process gets its own private memory space – which basically solves all four problems above.

- **Linear address space:** ordered set of contiguous non-negative integer addresses, {0,1,2,3, ... }
- **Virtual address space:** set of $N = 2^n$ virtual addresses, {0,1,2,3, ... , $N-1$}
- **Physical address space:** set of $M = 2^m$ physical addresses, {0,1,2,3, ... , $M-1$}

Every byte in main memory has one physical and one or more virtual address. This is managed by the **memory management unit (MMU)**, which also performs cache-checking.



The advantages of **virtual memory (VM)** are numerous: it makes more efficient use of the limited RAM available (with e.g. paging), it simplifies memory management for programmers, and isolates address spaces.

Some uses of virtual memory

1. **Caching:** blocks of memory called pages are stored on disk (as virtual pages) and cached in DRAM (as physical pages). DRAM is about ten times slower than SRAM while still being

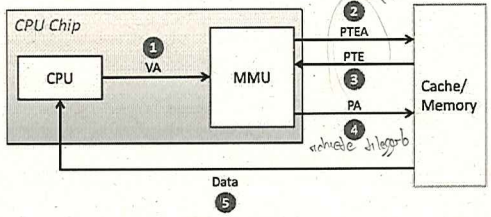
10,000 times faster than an HDD[29]. As a consequence, block sizes are rather big, the cache is fully associative with highly sophisticated and expensive replacement algorithms, and tends to be write-back. The reason this works is locality. The set of active virtual pages is called **working set**.

2. **Memory management**: each process has its own virtual address space and thus views memory as a simple linear array (yet in reality it's scattered all over the place). Each virtual page can be mapped to any physical page and can be stored in different physical pages at different times. This allows code and data to be shared among processes.
3. **Simplify linking and loading**: Linking: each program has a similar virtual address space (code, stack, and shared libraries always start at the same address). Loading: `execve()` allocates virtual pages for `.text` and `.data` sections (= creates PTEs marked as invalid). The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system.
4. **Memory protection**: extend PTEs with permissions bits which are checked by the page fault handler before remapping. If violated: SIGSEGV.

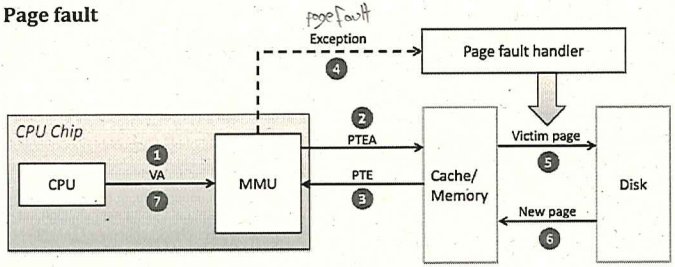
The address translation process

Page hit

1. Processor sends virtual address to MMU
2. MMU fetches PTE from page table in memory
3. Same as 2.
4. MMU sends physical address to cache/memory
5. Cache/memory sends data word to processor



Page fault

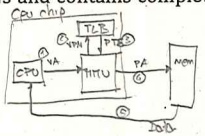


1. Processor sends virtual address to MMU
2. MMU fetches PTE from page table in memory
3. Same as 2.
4. Valid bit is zero, so MMU triggers page fault exception
5. Handler identifies victim (and, if dirty, pages it out to disk)
6. Handler pages in new page and updates PTE in memory
7. Handler returns to original process, restarting faulting instruction

[29] According to Prof. Hoefler fromt the mentioned "Operating Systems and Networks" course, this "maybe decreases by a factor of 10 for SSDs".

Translation lookaside buffers (TLB)

The TLB is a small hardware cache in the MMU and maps VPNs to PPNs and contains complete PTEs for a small number of pages.



A simple memory system example

Omitted. Please have a look at slides 35 ff.

Multi-level page tables

A virtual page may refer to a page-aligned region of the virtual address space *and* contents thereof. A physical page is a page-aligned region of physical memory. A physical **frame** (= physical page) is an alternative terminology: page = contents, frame = container.

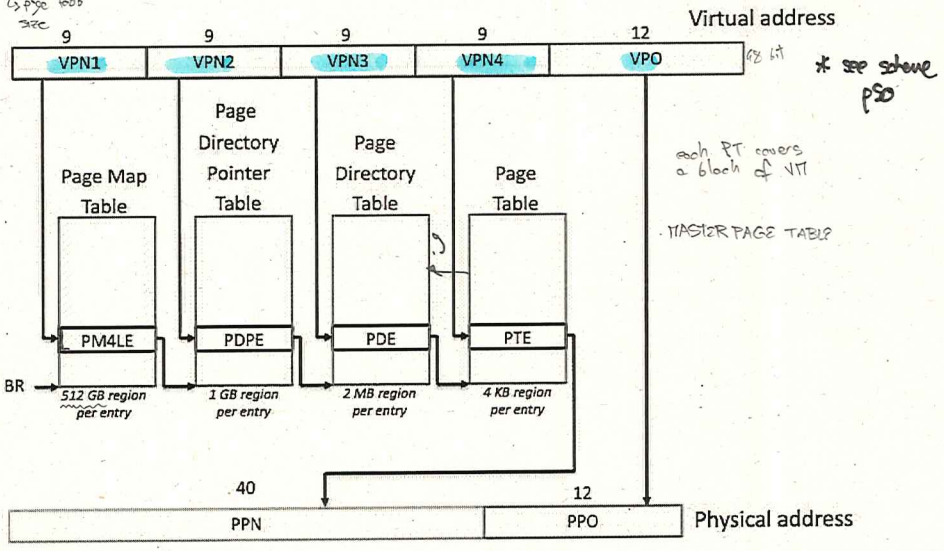
For a 4KB page size, a 48-bit address space (used on 64-bit machines) and 8B PTE, a page table of size $2^{48}\text{B}/2^{12}\text{B} \cdot 2^{3}\text{B} = 2^{39}\text{B} = 512\text{GB}$ is required. To circumvent that problem, multi-level page tables are used.

Case study of the Core i7 ™ virtual memory System

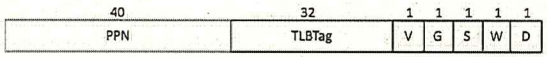
Partly omitted. See slides 49 ff.

Components of the virtual address (VA)	Components of the physical address (PA)
- TLBI: TLB index - TLBT: TLB tag - VPO: virtual page offset - VPN: virtual page number	- PPO: physical page offset (same as VPO) - PPN: physical page number - CO: byte offset within cache line - CI: cache index - CT: cache tag

Paging in x86-64 uses 48-bit virtual address (since 64 bits is a lot), 52-bit physical address (which equals 40 bits for PPN (4KB pages, $52 - \log 4096 = 40$)), and there are 512 entries per page (PT/PTE = 4096/8).



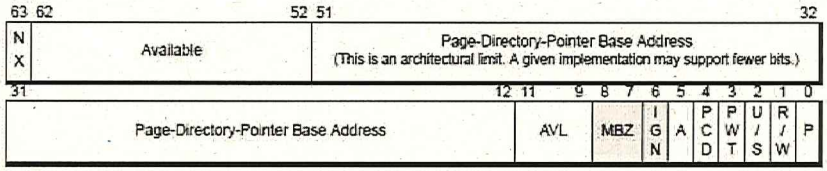
TLB entry: (Core i7)



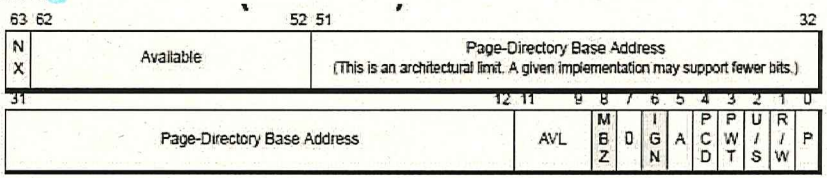
- **V:** indicates a valid (1) or invalid (0) TLB entry
- **TLBTag:** disambiguates entries cached in the same set
- **PPN:** translation of the address indicated by index & tag
- **G:** page is "global" according to PDE, PTE
- **S:** page is "supervisor-only" according to PDE, PTE
- **W:** page is writable according to PDE, PTE
- **D:** PTE has already been marked "dirty" (once is enough)

Page table entries: (skipped) *see slides for all details

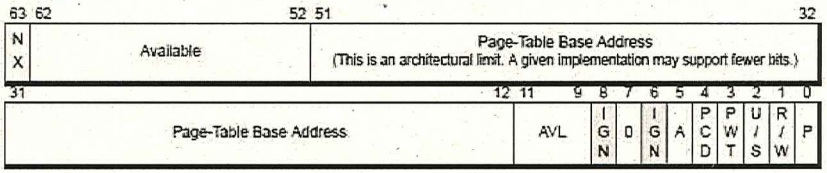
Level 4



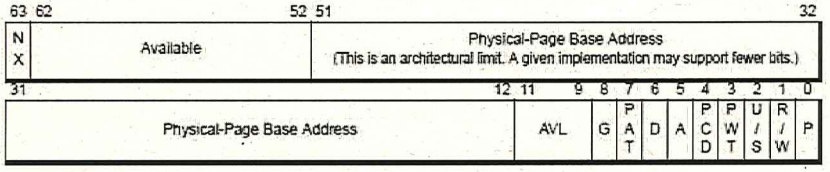
Level 3



Level 2



Level 1



Flags in PTEs:

- **Avail:** available for system programmers
- **G:** global page (don't evict from TLB on task switch)
- **PAT:** Page-Attribute Table
- **PCD:** cache disabled or enabled
- **PWT:** write-through or write-back cache policy for this page
- **U/S:** user/supervisor
- **D:** dirty (set by MMU on writes)
- **A:** accessed (set by MMU on reads and writes)
- **R/W:** read/write
- **P:** page is present in physical memory (1) or not (0)

Large pages

Large pages are created by concatenating VPN4 and VPO which results in 21 bits = 2MB pages. If VPN3 is also used for the concatenation there are 30 available bits which result in a 1GB page, called **huge** page.

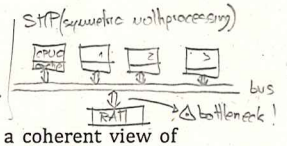
Large and huge pages have advantage of terminating the page table walk early and simplify address translation. They are useful for programs with very page, contiguous working sets (reduces compulsory TLB misses). (but it's hard to use)

Optimizing for the TLB

Omitted.

19 Multiprocessing

problem: 1 processor isn't fast enough
observation: some jobs can parallelize
solution: attach multiple processors to system bus



Consistency and Coherence

Coherency: values in caches all match each other and processors all see a coherent view of memory. **Consistency**: the order in which changes to memory are seen by different processors.

Program order: the order in which a program on a processor appears to issue reads and writes (even on a uniprocessor this isn't equal to the order the CPU issues these calls). This only refers to local reads/writes. **Visibility order**: the order which all reads and writes are seen by the processor(s). This refers to all operations in the machine. *Each processor reads the value written by the last write in visibility order.*

elaborate why
• DIG
• PPROG

On modern machines most CPU cores are cache coherent which means they behave as if they were all accessing a single memory array. This makes programming easier but is hard to implement and memory is slower as a result.

↳ other option is deferred, with dirty bit

Memory consistency (i.e. what value is read by each processor) is very important but even a simple statement like "last value written" is hard to implement which is the reason for many different models.

Sequential Consistency

1. Operations from a processor appear (to all others) in program order.
2. Every processor's visibility order is the same interleaving of all the program orders.

This requires each processor to issue memory operations in program order. The RAM has to have a total order on all operations and furthermore the operations have to be atomic.

Advantages	Disadvantages
- Easy to understand for the programmer - Easy to write correct code to - Easy to analyze automatically	- Hard to build a fast implementation - Cannot reorder reads/writes, not even in the compiler and not even from a single processor! - Cannot combine writes to same cache line (write buffer) - Serializing ops at memory controller is too restrictive (see NUMA)

MSI
MESI
MOESI
MESIF

Cache coherence with snooping (wiretap)

The cache snoops on reads/writes from other processors. If a line is valid in the local cache and another processor writes to that line, the local line is invalidated. This requires a write-through cache. The line can be valid in many caches until a write happens.

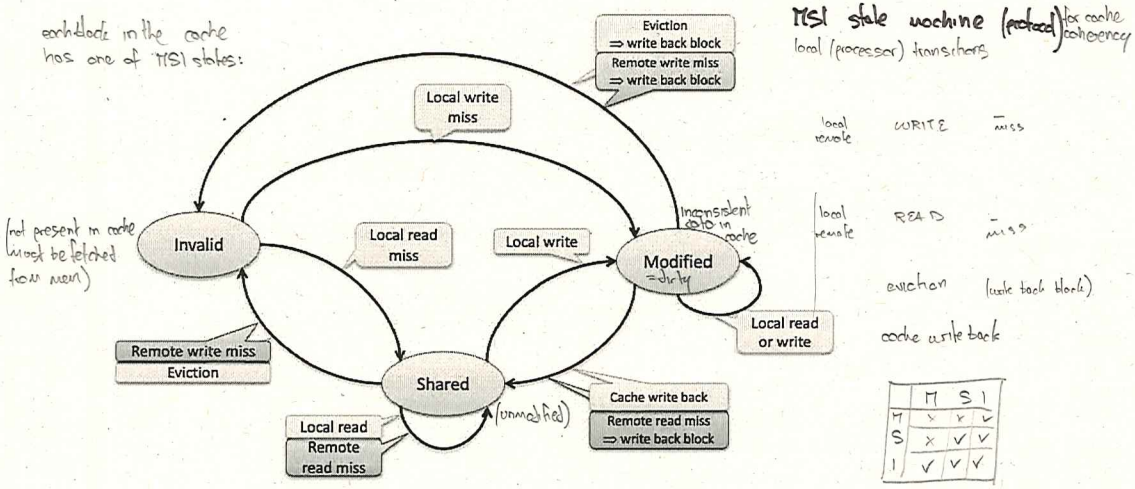
→ write immediately

Should the cache be write-back, the lines can have an additional "dirty" (modified) state, at maximum in one cache. This requires a cache coherency protocol such as **MSI** – Modified, Shared, and Invalid. The cache logic responds to processor/remote bus reads/writes, change cache line state, and write back data (**flush**) if required.

each block in the cache has one of MSI states:

MSI state machine (protocol) for cache coherency

local (processor) transitions



local remote WRITE miss

local remote READ miss

eviction (write back block)

cache write back

	M	S	I
M	x	x	✓
S	x	✓	✓
I	✓	✓	✓

Unfortunately, MSI has a few issues. In the "I" state when a write miss is executed it first needs to read the line and if someone else hast it in "M" state, it needs to wait for the flush to happen. When in the "M" state and another core observers a remote read the line has to be flushed (obviously) but now there's an issue with invalidation: should the line be invalidated yet it was supposed to be shared, there's an extra miss. When transitioning to a shared state however, it might be remote write miss which then causes an extra invalidate.

The MESI cache coherence protocol

MESI has four states, a new signal, and a new bus signal:

- **Modified**: only copy, dirty
- **Exclusive**: only copy, clean
- **Shared**: several copies, all clean
- **Invalid** (not in cache)
- **HIT**: signal to remote processor its read hit in the local cache
- **RdX** (read exclusive): cache can load into either "S" or "E" state. Other caches can see the type of read.

Observations:

- Dirty data is always written though memory, there are no cache-cache transfers which makes it an invalidation-based protocol
- Data is always either dirty in *one* cache (needs to be written back before a remote read) or clean (can be safely fetched from memory).
- MESI is good if the memory latency is a lot smaller than the latency of a remote cache, since it adds 5 memory accesses to get rid of the ambiguity of the remote read/write misses.

	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	N

- Caches don't have to be able to distinguish between remote read/write misses (issue RdX)

Terminology:

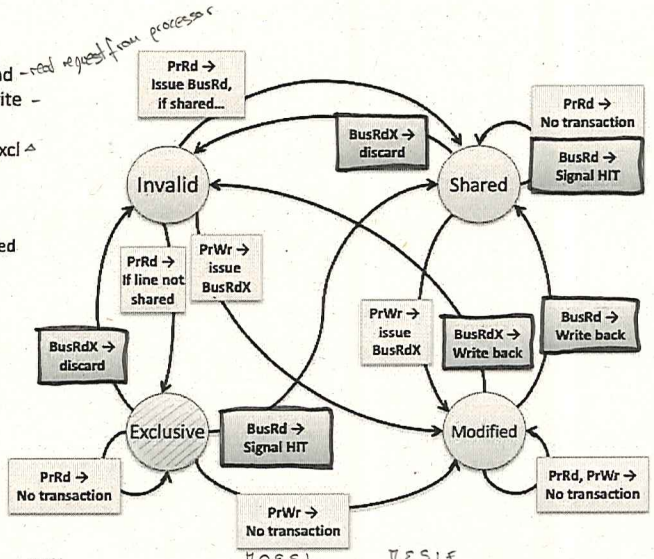
- PrRd: processor read – read request from processor
- PrWr: processor write –
- BusRd: bus read *
- BusRdX: bus read excl Δ
- BusWr: bus write

Processor-initiated

Snoop-initiated (keep an eye on)

*read request from bus, no intent to modify

Δ with intent



AMD INTEL

Partly omitted (MOESI, MESIF). (owner, forward)

MOESI

	M	O	E	S	I
M					✓
O				✓	✓
E					✓
S		✓		✓	✓
I	✓	✓	✓	✓	✓

MESIF

	M	E	S	I	F
M				✓	
E				✓	
S			✓	✓	✓
I	✓	✓	✓	✓	✓
F			✓	✓	

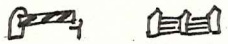
Relaxing sequential consistency

Ways to relax SC include:

- Write-to-read: later reads can bypass earlier writes
- Write-to-write: later writes can bypass earlier writes
- Break write atomicity (no single visibility order)
- Weak ordering: no implicit order guarantees at all

Partly omitted (processor-specific ways to relax SC).

Barriers and fences



Generally, the weaker the consistency model is, the faster it goes in hardware. Visibility order is essential for the correct functioning yet difficult to guarantee (compilers, memory models). **Barriers** (aka **fences**) are a very good solution.

- Compiler barriers: prevent compiler reordering statements
- Memory barriers: prevent CPU reordering instructions

Multicore synchronization: Test-and-Set

TAS

One of the simplest non-trivial atomic operations

1. Read a memory location's value into a register
2. Store "1" into the location

This requires a read-modify-write cycle (i.e. memory bus has to "locked" during instruction). It can also appear as a register.

TAS can be very expensive since the memory has to be locked while a long operation occurs, it has to do a read, followed by a write while no-one else can access memory, and if it is spinning, it slows things down. As a alternative, "Test and Test-and-Set" can be used *but* to be able to use it, the programmer needs to understand a lot of the inner workings of a lot of things. Simply put: don't use it, just don't.[30]

```
void acquire(int *lock) {
  do {
    while (*lock == 1);
  } while ( TAS(lock) == 1);
}
```

Compare and Swap

CAS can implement almost all wait-free data structures for which it requires bus locking (or similar) in the memory system.

1. Load `location` into `value`
2. If `value == old` then store `new` to `location`
3. Return `value`

The general pattern where CAS is used is read-copy-update where writers take a copy, modify it, then write back the copy. The old version is deleted when all the readers are finished.

Since CAS reports when a single location is different but does not report when it is written (with the same value), it suffers from the ABA problem. To solve this, the values has to change *always*. This is done by splitting the value into the original value and a monotonically increasing counter. (Yet, everything can be done fast with CAS, if you're slightly clever.)

The "ABA" problem:

1. CPU A reads value as x
2. CPU B writes y to value
3. CPU B writes x to value
4. CPU A reads value as x

⇒ concludes nothing has changed

Simultaneous multithreading

Cache-coherent **SMP** still has the memory as its bottleneck – all accesses to main memory stall the processor (even with MOESI, which allow reads to be serviced from another cache). And memory stalls halt the processor, also other processors which access memory. During these times when the processor is waiting for memory/another cache, most functional units are idle and many instructions do not require the memory unit, yet **ILP** is limited due to data dependencies. This leads to the idea of executing instructions in other threads (there are multiple fetch/decode units and registers) to reuse superscalar functional units. This is done by labelling instructions in hardware with a thread id. A CPU which does this appears as multiple CPUs to the OS. The benefit is around 10-20% but since it's cheap (transistor-wise) to implement, it's worth it, especially for lots of memory-intensive requests (not so much scientific computing though).

Non-Uniform Memory Access (NUMA)

NUMA removes the memory bottleneck by having multiple, independent memory banks to which processors have independent paths. The interconnect is not a bus anymore but a network link (which passes messages). All memory is globally addressable but local memory is faster.

NUMA cache coherence

To ensure cache coherency in a NUMA environment, either the bus is emulated (similar to snooping but with messages) or a **cache directory** is used, where each entry consist of the cache line,

[30] Just quoting slide 47...

the owner, and one bit per node indicating presence of this line in that node. This si useful when lines are not widely shared and when there are a lot of NUMA nodes since it reduces interconnect traffic and load at each node yet it requires lots of fast memory.

Performance implications of multicore

- Memory latency
- Cache access latency
- False sharing

Optimization example: MCS locks

Omitted.

20 Devices

Some parts of this chapter were copied shamelessly from http://studysheets.ch/sheets/operating-systems/download. Inspiration came from the same behavior by Prof. Roscoe and Prof. Hoefler.

To an OS programmer, a device is a piece of hardware visible from software occupying some location on a **bus**. It also has a set of **registers** (which are memory mapped or in IO space) and is a source of **interrupts**. It also may imitate **Direct Memory Access** transfers.

Device registers

CPU can load from device registers:

- Obtain status info
- Read input data

CPU can store to device registers:

- Set device state and configuration
- Write output data
- Reset states

Registers can be addressed in different ways: memory mapped[31], using I/O instructions, or using indirection (to save I/O space). These registers do not behave like RAM since they might change without writes from the CPU and writes to these registers are used to trigger actions (e.g. send data, reset state machine ...).

The details of **registers** are given in chip "datasheets" or "data books" (this information is rarely trusted by OS programmers). A very simple UART (Universal asynchronous receiver/transmitter) driver might be using programmed IO (**PIO**):

- CPU explicitly reads and writes all values to and from registers
- All data must pass through CPU registers

And uses polling:

- CPU polls device register waiting before send/receive
- Can't do anything else in the meantime
- Without CPU polling, no I/O can occur

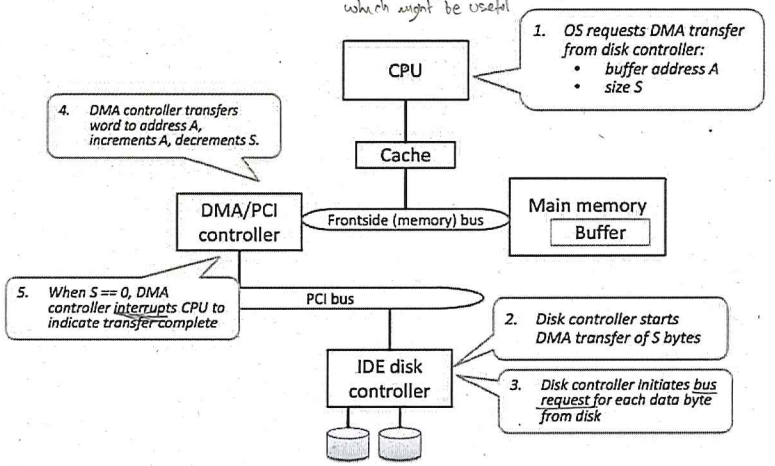
[31] cf. Digital Circuits class MIPS programming

Dealing with caches

Dive register access *must* bypass the cache (PTEs have the "no cache" flag set) to avoid possible inconsistencies caused by non-CPU writes. Additionally write-back caches and write buffers can cause problems, and read and writes cannot be combined into cache lines.

Direct Memory Access

Direct Memory Access is used to avoid programmed I/O for lots of data (e.g. fast network or disk interfaces). This requires a DMA controller (which is generally built-in) and this bypasses the CPU to transfer data directly between I/O device and memory, thus not taking up CPU time and might save memory bandwidth and there's only one interrupt per transfer. There's no need for the CPU to copy data which in turn does not pollute its cache. Memory can be accessed on demand and there is a performance gain because CPU and device work in parallel.


CPU
1. OS requests DMA transfer from disk controller:
• buffer address A
• size S
Cache
4. DMA controller transfers word to address A, increments A, decrements S.
DMA/PCI controller
Frontside (memory) bus
Main memory
Buffer
5. When S == 0, DMA controller interrupts CPU to indicate transfer complete
PCI bus
2. Disk controller starts DMA transfer of S bytes
IDE disk controller
3. Disk controller initiates bus request for each data byte from disk

Caches

Due to DMA memory becomes inconsistent with CPU caches. This leaves these options:

- CPU can map DMA buffers non-cacheable → large hit, probably wants to process data anyway
- Cache can "snoop" DMA bus transactions (but doesn't scale beyond small SMP systems)
- OS can explicitly flush/invalidate cache regions → cache management important part of device drivers

Virtual memory

DMA addresses are physical yet OS and user mostly deal with virtual addresses. This requires address translation, possibly more than just a hardware page table due to non-contiguity of the physical address space. Newer systems provide an **IOMMU**, which does the same for the I/O devices as MMU does for the CPU.

Device drivers

Driver and device are both state machines which need data to be transferred between each other and signals are used to signal state transitions. There are four states: write a device register, read a device register, device request interrupt, and shared memory (which is the only asynchronous state).

Buffer rings and descriptor rings

The ring consists of either buffers in contiguous memory or pointers (**descriptors**) to other bits of memory. OS and device pointers move independently around the ring. This provides a buffer of packets and requires very little explicit coordination. Most modern devices deal with buffer descriptors which offer – via a level of indirection – pointer area(s) of memory and metadata. This allows software more flexible data placement, variable sized buffers which can vary dynamically. Additionally, this does not require data and metadata to be mixed.

What happens when one pointer catches up with the other? (**overruns** and **underruns**; this corresponds to producer-consumer queues using messages and interrupts and not by using mutexes/monitors/condition variables/threads)

Transmit

Device has no more packets to send → it must wait

- Could continue to poll memory until next descriptor is owned by it
- Could go to sleep and signal the software to wake it up (monitor)

CPU has no more slots to send packets → must wait

- Can spin polling, but inefficient
- Signals device to interrupt it when a packet has been sent i.e. a buffer slot is now free

Receive

Device has no buffers for received packets → starts discarding packets

- Not as bad as it sounds
- Will start copying them to memory when a buffer is free
- Signals that it's lost some in a status register

CPU reads all received packets → it must wait

- Can spin polling, but inefficient
- Signals device to interrupt it when a new packet has been received
- Goes off to do something else

More complex devices

Omitted.

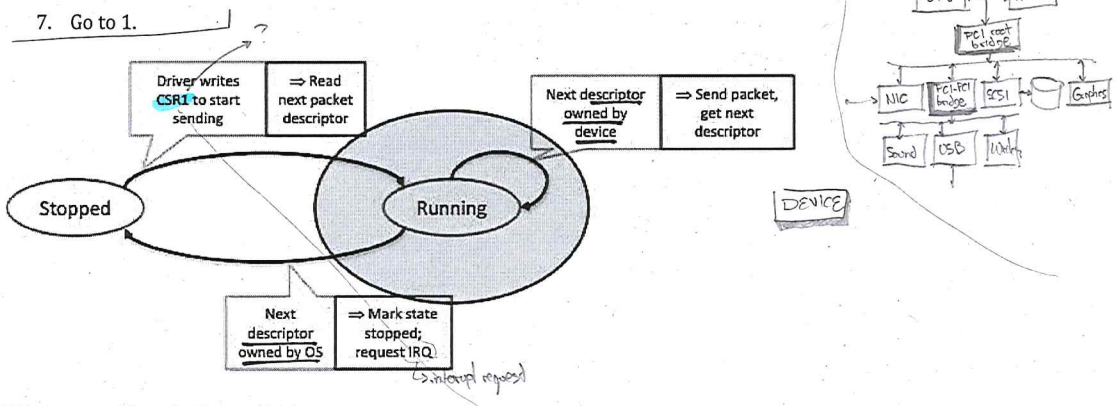
Device initialization

To initialize a device, the system and the device need to ensure state transitions are synchronized, which is done as follows:

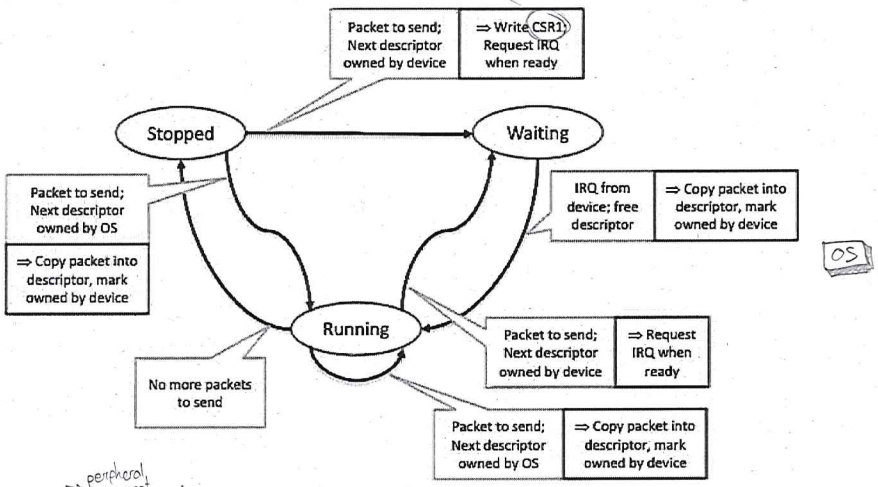
1. Wait for the hardware to settle down
2. Stop the device doing anything, just to be sure: no interrupts, no DMA, no sending packets
3. Create shared data structures: e.g. descriptor rings, must tell device where they are!
4. Write registers to start device running

I/O state machines (hardware side)

1. DMA Read: descriptor
2. If `descriptor.owner == "OS"` then enter state "stopped"
3. DMA Read: buffer
4. Send packet
5. DMA Write: `descriptor.owned ← "OS"`
6. Calculate next descriptor address: next in memory (for unchained) or value of buffer field 2 (for chained mode)



I/O state machines (software side)


Packet to send; Next descriptor owned by device
⇒ Write CSR1; Request IRQ when ready
Stopped
Waiting
Packet to send; Next descriptor owned by OS
⇒ Copy packet into descriptor, mark owned by device
IRQ from device; free descriptor
⇒ Copy packet into descriptor, mark owned by device
OS
Running
Packet to send; Next descriptor owned by device
⇒ Request IRQ when ready
No more packets to send
Packet to send; Next descriptor owned by OS
⇒ Copy packet into descriptor, mark owned by device

Since PCI-based DMA transfers are only coherent with CPU caches on x86, the following needs to be implemented:

	DMA reads	DMA writes
Before	CPU should flush the cache for that address → main memory is up to date	CPU should flush or invalidate cache → no dirty lines left to write to memory
After	CPU should invalidate cache for that address → cache doesn't hold old value	CPU invalidates cache → cache doesn't hold old value

Discoverable busses: PCI

Peripheral Component Interconnect (PCI) is an electrical standard for connecting devices., a standard for physical connectors, a set of bus protocols for inter-device communication, and a software-visible interface to I/O hardware. It tries to solve many problems:

- **Device discovery**: finding out which devices are in the system
- **Address allocation**: which addresses should each device's registers appear at?

- **Interrupt routing**: which interrupt signals from the device should map to which exception vectors?
- **Intelligent DMA**: "bus mastering" devices no longer need a DMA controller

The connections are represented as a tree, the address space is flat. PCI devices are self-describing. To find all the devices, you first must find the PCI "root complex" bridge atop of the tree. Then the configuration is read to find all attached devices, add them to the list of devices and functions, and record requirements for address space – if it is a bridge, recurse. This results in a list of all devices, complete with their address space requirements.

To allocate addresses, find address ranges for each device and bridge

Requirements include:

- Each device has the size of address ranges it needs
- All devices "below" a bridge have ranges that fit into the bridge's range
- Each bridge has a range which includes all it's "children".
- Each range is aligned to some power-of-2 boundary

When these requirements are found, the following needs to be programmed:

- Each PCI bridge with translation information
- Each device with "base-address/range" (BAR) registers

PCI interrupts

Four interrupt lines

- INTA, INTB, INTC, INTD...
- Bridges allow arbitrary wiring of device lines to bridge lines
- Translated by root bridge into system interrupt

PCI Express introduces MSIs

- Message-signaled interrupts
- Interrupt encoded as PCI write to specified address range
- Translated by root bridge into system interrupt
- Interrupts can be individually steered to particular cores/APICs

PCI allows **bus mastering**: a device can issue read/write transactions to anywhere in memory, even to other PCI devices. This makes external DMA controllers obsolete since the controller is effectively integrated with the device itself while still following the principle of the device DMA-ing data to/from memory. This gives much more flexibility and allows for more intelligence.

A quick look at the future

Omitted.

Sources

Unless otherwise noted: Lecture slides by Timothy Roscoe available on the course website accompanying the course 252-0061-00L taught in the fall semester 2015 at ETH Zürich. That ETH course in turn is partly based on CS 15-213 at Carnegie Mellon University and CSE333 at the University of Washington. Simple definitions might be from Wikipedia.